

版 权 声 明

Original edition, entitled *Algorithm Design: Foundations, Analysis, and Internet Examples*, by Michael T. Goodrich and Roberto Tamassia, ISBN 0-471-38365-1, published by Wiley Publishing, Inc. Copyright © 2002 John Wiley & Sons, Inc.

All rights reserved. This translation published under license.

Translation edition published by POSTS & TELECOM PRESS Copyright 2006.

Java[®] is a trademark of Sun Microsystems, Inc.

Unix[®] is a registered trademark in the United States and other countries, licensed through X/Open Company, Ltd.

All other product names mentioned herein are the trademarks of their respective owners.

本书简体中文版由Wiley Publishing, Inc.授权人民邮电出版社独家出版。

版权所有，侵权必究。

译者序

本书是算法分析与设计方面的优秀著作。它系统地阐述了算法设计的方法、技术和应用实例。书的内容具有一定的深度和广度，包括基础算法、基本数据结构、基本算法设计技术、图算法、网络流和匹配、文本处理算法、数论算法、网络算法、NP完全性、近似算法、回溯法和分枝限界法、外存算法、并行算法和在线算法。

算法设计一直是备受广泛关注的研究主题。这本教科书介绍了算法设计研究领域的最新进展。因特网示例展示了传统算法在因特网领域中的应用。

本书分算法基础（第1章～第5章）、图算法（第6章～第8章）、因特网算法（第9章～第11章）和高级主题（第12章～第14章）四个部分。本书主要内容及特点如下：

- 把数据结构和算法设计方法与应用实例紧密联系在一起。相关例子包括栈在Web浏览器中的应用，集合求交集算法在因特网搜索引擎中的应用、图模型在面向对象程序及因特网中的表示。
- 因特网中的大量问题促进了新算法的研究和传统算法的应用。相关的例子包括信息检索、Web爬行、数据分组路由、Web拍卖算法和Web高速缓存算法。通过因特网应用算法主题大大提高了学生对学习算法的兴趣。
- 在文本处理一章中全面介绍了涉及trie的各种数据结构，以及它们在搜索引擎中的应用。
- Java实现示例，覆盖了软件设计方法、面向对象实现问题和算法的实验性分析。这些典型问题的Java应用示例分布在不同的章节中。
- 以深入浅出的方式介绍了NP完全性理论，引入了P类问题和NP类问题的定义。通过网络路由器最优配置问题、网络服务器带宽优化问题和因特网站点拍卖问题等现实中的具体问题，说明为什么要研究NP完全问题。同时给出了许多重要的NP完全问题的实例。讨论了近似算法在NP完全问题中的应用。
- 书中习题分为三类，一类是基础题，用于巩固所学概念和方法，另一类是需要一些综合知识才能求解的创新题，最后一类是结合所学方法和应用问题的程序设计题。
- 本书以大量图例说明算法的工作过程，使得算法更加易于理解和掌握。

由于时间紧迫及译者水平有限，译文难免有错误及不妥之处，恳请读者批评指正。

霍红卫
西安电子科技大学计算机学院
2006年8月

前言

本书旨在全面介绍计算机算法设计与分析和数据结构的内容。依据计算机科学和计算机工程的课程设置，本书主要适用于大学三年级或四年级，或者某些学校一年级研究生的算法课程。

涵盖的主题

本书涵盖了离散算法设计与分析领域的内容，取材广泛，包括下面几个方面。

- **算法设计与分析**，包括渐近表示、最坏情况分析、平摊分析、随机分析和实验分析。
- **算法设计方法**，包括贪心法、分治法、动态规划、回溯法和分枝限界法。
- **算法框架**，包括NP完全性、近似算法、在线算法、外存算法、分布式算法和并行算法。
- **数据结构**，包括表、向量、树、优先队列、AVL树、(2, 4)树、红黑树、伸展树、B树、散列表、跳跃表、合并寻找树。
- **组合算法**，包括堆排序、快速排序、归并排序、选择、并行表排列和并行排序。
- **图算法**，包括遍历（DFS和BFS）、拓扑排序、最短路径（所有点对最短路径和单源点最短路径）、最小生成树、最大流、最小代价流和匹配问题。
- **几何算法**，包括范围查找、凸包、线段相交和最近点对问题。
- **数值算法**，包括整数相乘、矩阵相乘和多项式相乘、快速傅里叶变换（FFT）、扩展的Euclid算法、模求幂和素性测试。
- **因特网算法**，包括分组路由、多播、领导人选举、加密、数字签名、文本模式匹配、信息检索、数据压缩、Web高速缓存和Web拍卖。

致教师

本书主要用作高年级算法课程的教科书，也可用作某些学校一年级研究生算法课程的教材。本书习题量大，可分为基础题、创新题和程序设计。本书专门针对教学做出了如下安排：

- **可视化证明**（即图形化证明），使学生更易于理解数学公式。可视化证明的一个例子是分析自底向上构造堆的过程。传统上这部分内容对于学生而言是很难理解的，因此教师要花相当多的时间进行解释。本书包含的可视化证明直观、严谨和快速。
- **算法设计方法**，提供了算法设计和实现的一般性技术。例子包括分治法、动态规划、修饰模式和模板方法模式。
- **使用随机技术**，算法中利用随机选择简化其设计与分析。这种用法用对简单数据结构和算法的直观分析，代替了复杂数据结构的复杂平均情况分析。例子包括跳跃表、随机化快速排序、随机化快速选择和随机化素性测试。
- **因特网算法主题**，既从新的因特网观点激发了传统算法主题，同时又强调因特网的应用引出的新算法。例子包括信息检索、Web爬行、分组路由、Web拍卖算法和Web高速缓存算法。通过因特网应用算法主题大大提高了学生对学习算法的兴趣。
- **Java实现示例**，覆盖软件设计方法、面向对象实现问题和算法的实验性分析。这些实现

示例分布在不同的章节中。教师可以从中进行选取，既可在课堂上讲授，又可作为附加阅读材料，也可以一带而过。

本书结构合理，给予教师充分的自由来组织和介绍内容。同样，每章之间的相关性相当灵活，教师可以定制算法课程，突出他们认为重要的部分。我们还广泛讨论了因特网算法主题，结果表明学生对这部分内容相当感兴趣。此外，在许多地方还包括了传统算法在因特网中应用的示例。

表0-1说明了如何在传统算法导论课程中使用本书，其中有些主题是因特网应用引出的。

表0-1 传统算法导论课程的大纲计划，每章包括可选部分

章	主 题	选 讲
1	算法分析	实验分析
2	数据结构	堆Java示例
3	查找	3.2~3.5节中的一节
4	排序	原位快速排序
5	算法技术	FFT算法
6	图算法	DFS算法Java示例
7	加权图	Dijkstra算法Java示例
8	匹配和流	课程结束时包括
9	文本处理（至少一节）	trie
12	计算几何	课程结束时包括
13	NP完全性	回溯
14	框架（至少一节）	课程结束时包括

本书还可专门用于因特网算法课程，它回顾了传统算法主题，但这是从因特网的角度进行的，同时还涵盖了源于因特网应用的新的算法主题。表0-2说明了如何将本书用于因特网算法课程。

表0-2 因特网算法课程的大纲计划，每章包括可选部分

章	主 题	选 讲
1	算法分析	实验分析
2	数据结构（包括散列法）	快速回顾
3	查找（包括3.5节，跳跃表）	搜索树Java示例
4	排序	原位快速排序
5	算法技术	FFT算法
6	图算法	DFS算法Java示例
7	加权图	跳过一个MST算法
8	匹配和流	匹配算法
9	文本处理	模式匹配
10	安全和密码学	Java示例
11	网络算法	多播算法
13	NP完全性	课程结束时包括
14	框架（至少2节）	课程结束时包括

当然也可以有其他选择，比如，将传统算法课程和因特网算法课程结合起来。对此，这里不做过多讨论，感兴趣的教师可以进行独创性的安排。

网上增值教辅

本书配套网站如下：

<http://www.wiley.com/college/goodrich>

这个网站包含大量教辅信息，增加了本书的主题。特别为学生提供了如下信息：

- 本书大部分主题的演示文稿课件（一页4张幻灯片的格式）。
- 所选作业的提示数据库，并按题号索引。
- 交互式Java小程序（applet），动态模拟基本数据结构和算法。
- 本书Java示例的源代码。

我们感觉其中的提示应该是很有趣的，尤其是对于创新题，这些习题对于某些学生具有相当大的挑战性。

对于使用本书作为教材的教师，本网站还专为其提供了一部分内容，其中包括如下教辅材料：

- 本书中的部分习题的答案。
- 附加习题及其答案的数据库。
- 本书大部分主题的演示文稿课件（一页1张的格式）。

对数据结构和算法实现感兴趣的读者可从<http://www.jdsl.org>下载JDSL，即 *Data Structure Library in Java*。

预备知识

本书假定读者掌握基本数据结构，如数组和链表，并且至少熟悉一种高级程序设计语言，如C、C++或Java。不过，书中所有算法均用高级伪代码描述，特定的程序设计语言仅在所选的Java实现示例中使用。

关于数学背景，假设读者熟悉大学一年级的数学课程，包括指数、对数、求和、极限和基本概率知识。即便如此，我们仍在第1章中回顾了其中大部分数学知识，包括指数、对数和求和，并且在附录A中总结了其他有用的数学知识，包括概率基础。

致谢

许多人对本书给予了帮助。特别要感谢Jeff Achter、Ryan Baker、Devin Borland、Ulrik Brandes、Stina Bridgeman、Robert Cohen、David Emory、Devid Ginat、Natasha Gelfand、Mark Handy、Benoît Hudson、Jeremy Mullendore、Daniel Polivy、John Schultz、Andrew Schwerin、Michael Shin、Galina Shubina和Luca Vismara。

非常感谢我们以前的助教帮助开发了习题、程序设计作业和算法动画系统。许多朋友和同事对本书给出了有益的评述，使本书不断完善。我们特别要感谢Karen Goodrich、Eugene Luks、Art Moorshead和Scott Smith富有洞察力的评论。同时还要真挚感谢那些不知名的审阅人给予的详细而又富有建设性的批评，他们的评论非常有用。

感谢编辑Paul Crockett和Bill Zobrist对于本书给予的热情支持。Wiley的产品组一直非常优秀。还要感谢那些在本书的编写过程中给予帮助的人。他们是Susannah Barr、Katherine Hepburn、Bonnie Kubat、Sharon Prendergast、Marc Ranger、Jeri Warner和Jennifer Welter。

书稿中的文本主要用 \LaTeX 完成，图片用Adobe FrameMaker[®]和Visio[®]完成。LGrind系统用于

将Java代码段格式化成 \LaTeX 。CVS版本控制系统理顺了我们（有时并发）的文件编辑工作。

最后，我们还要热情地感谢Isabel Cruz、Karen Goodrich、Giuseppe Di Battista、Franco Preparata、Ioannis Tollis和我们的父母，他们在本书的准备过程中一直在给我们提供建议、鼓励和支持。同时感谢他们提醒我们在写作之余，好好地享受生活。

Michael T. Goodrich
Roberto Tamassia

作者简介

Goodrich教授和Tamassia教授是数据结构和算法领域的著名学者，在该领域发表了涉及因特网计算、信息可视化、地理信息系统和计算机安全等方面的多篇论文。他们的研究得到美国国家自然科学基金、陆军研究办公室以及美国国防部高级研究计划署的支持。同时在教育技术领域，他们在算法可视化系统和支持远程教育的基础设施方面也相当活跃。

Michael Goodrich于1987年从普度大学获得计算机科学博士学位，目前是加州大学（欧文分校）信息和计算机科学学院计算机科学系的教授，网络安全与隐私中心主任。此前，他是约翰·霍普金斯大学计算机科学系的教授，同时也是霍普金斯算法工程中心的主任。Goodrich教授是*International Journal of Computational Geometry & Applications*、*Journal of Computational and System Sciences*和*Journal of Graph Algorithms and Applications*等学术刊物的编委。

Roberto Tamassia于1988年从伊利诺伊大学（Urbana-Champaign）获得电子和计算机工程博士学位，目前他是布朗大学计算机科学系的教授和几何计算中心的主任。Tamassia教授是*Computational Geometry: Theory and Applications*、*Journal of Graph Algorithms and Applications*的编委，曾经担当过*IEEE Transactions on Computers*的编委。

除了在研究方面所取得的成就之外，两位作者在教学方面也经历颇丰。自1987年以来，Goodrich博士一直从事数据结构和算法方面的教学工作，为大学一、二年级学生开设数据结构课程，为高年级学生开设算法导论课程，并多次获得教学成果奖。他的教学风格生动，通过课堂上和学生互动，使学生对于数据结构和算法设计技术既有直观的了解，又有相当深入的认识，同时在进行算法分析时又不失数学的严谨。自1988年以来，Tamassia博士一直为大学低年级学生开设数据结构和算法导论课程。他在布朗大学为研究生开设的计算几何课程广受学生的好评，也吸引了众多的学生，包括本科生。在教学中，继续沿用布朗大学“电子课堂”的传统，还有效地利用交互式超媒体演示，使其教学风格独树一帜。由Tamassia博士精心设计的课程网页已经广泛用作全世界的学生和专业人员的参考材料。

目 录

第一部分 基础工具

第1章 算法分析.....2

1.1 算法的分析方法学.....2	
1.1.1 伪代码.....4	
1.1.2 随机存取机(RAM)模型.....5	
1.1.3 统计基本操作的数量.....6	
1.1.4 递归算法分析.....7	
1.2 渐近符号.....8	
1.2.1 大O符号.....8	
1.2.2 与大“O”相关的渐近符号.....10	
1.2.3 渐近表示的重要性.....12	
1.3 数学概览.....13	
1.3.1 求和.....13	
1.3.2 对数和指数.....14	
1.3.3 简单证明技术.....16	
1.3.4 概率基础.....18	
1.4 算法分析案例研究.....20	
1.4.1 二次时间前缀平均值算法.....21	
1.4.2 线性时间前缀平均值算法.....22	
1.5 平摊方法.....22	
1.5.1 平摊技术.....23	
1.5.2 扩展数组实现分析.....25	
1.6 实验.....28	
1.6.1 实验组织.....28	
1.6.2 数据分析和可视化.....29	
1.7 习题.....31	
基础题.....31	
创新题.....33	
程序设计.....35	
1.8 本章注记.....36	

第2章 基本数据结构.....37

2.1 栈和队列.....37	
2.1.1 栈.....37	
2.1.2 队列.....40	
2.2 向量、表和序列.....43	

2.2.1 向量.....43	
2.2.2 表.....44	
2.2.3 序列.....48	
2.3 树.....49	
2.3.1 树抽象数据类型.....51	
2.3.2 树的遍历.....52	
2.3.3 二叉树.....55	
2.3.4 表示树的数据结构.....60	
2.4 优先队列和堆.....62	
2.4.1 优先队列抽象数据类型.....63	
2.4.2 PQ排序、选择排序和插入排序.....64	
2.4.3 堆数据结构.....66	
2.4.4 堆排序.....71	
2.5 字典与散列表.....76	
2.5.1 无序字典ADT.....76	
2.5.2 散列表.....77	
2.5.3 散列函数.....78	
2.5.4 压缩映射.....80	
2.5.5 冲突处理模式.....80	
2.5.6 通用散列.....83	
2.6 Java示例：堆.....85	
2.7 习题.....87	
基础题.....87	
创新题.....89	
程序设计.....91	
2.8 本章注记.....91	

第3章 查找树和跳跃表.....93

3.1 有序字典和二叉查找树.....94	
3.1.1 有序表.....94	
3.1.2 二叉查找树.....96	
3.1.3 二叉查找树中的查找.....96	
3.1.4 二叉查找树中的插入.....98	
3.1.5 二叉查找树中的删除.....99	
3.1.6 二叉查找树的性能.....100	
3.2 AVL树.....101	
3.2.1 更新操作.....102	

2 目 录

3.2.2 性能	105	创新题	171
3.3 深度有界查找树	106	程序设计	172
3.3.1 多路查找树	106	4.10 本章注记	173
3.3.2 (2,4)树	108	第5章 基本技术	174
3.3.3 红黑树	113	5.1 贪心法	174
3.4 伸展树	123	5.1.1 背包问题	175
3.4.1 伸展	123	5.1.2 任务调度	176
3.4.2 伸展过程的平摊分析	128	5.2 分治法	177
3.5 跳跃表	131	5.2.1 分治递归方程	177
3.5.1 查找	132	5.2.2 整数相乘	182
3.5.2 更新操作	133	5.2.3 矩阵相乘	183
3.5.3 跳跃表的概率分析	135	5.3 动态规划	185
3.6 Java示例: AVL树和红黑树	136	5.3.1 矩阵链乘	185
3.6.1 AVL树的Java实现	138	5.3.2 一般技术	187
3.6.2 红黑树的Java实现	141	5.3.3 0-1背包问题	188
3.7 习题	143	5.4 习题	190
基础题	143	基础题	190
创新题	144	创新题	191
程序设计	145	程序设计	191
3.8 本章注记	146	5.5 本章注记	192
第4章 排序、集合和选择	147	第二部分 图算法	
4.1 归并排序	147	第6章 图	194
4.1.1 分治法	148	6.1 图抽象数据类型	194
4.1.2 归并排序和递归方程	151	6.2 图的数据结构	199
4.2 集合抽象数据类型	152	6.2.1 边表结构	199
4.2.1 简单的集合实现	152	6.2.2 邻接表结构	201
4.2.2 具有union-find操作的划分	153	6.2.3 邻接矩阵结构	203
4.2.3 基于树的划分实现	154	6.3 图的遍历	204
4.3 快速排序	159	6.3.1 深度优先查找	204
4.4 基于比较的排序下界	162	6.3.2 双连通分量	206
4.5 桶排序和基数排序	163	6.3.3 广度优先查找	211
4.5.1 桶排序	163	6.4 有向图	213
4.5.2 基数排序	164	6.4.1 遍历有向图	214
4.6 比较排序算法	165	6.4.2 传递闭包	216
4.7 选择	166	6.4.3 DFS和垃圾收集	218
4.7.1 剪枝-查找法	166	6.4.4 有向无环图	219
4.7.2 随机化快速选择	166	6.5 Java示例: 深度优先查找	222
4.7.3 随机化快速选择分析	167	6.5.1 修饰模式	222
4.8 Java示例: 原位快速排序	168	6.5.2 DFS引擎	223
4.9 习题	170		
基础题	170		

6.5.3 模板方法设计模式	224	8.6 习题	281
6.6 习题	227	基础题	281
基础题	227	创新题	281
创新题	228	程序设计	282
程序设计	229	8.7 本章注记	282
6.7 本章注记	229		
第7章 加权图	230	第三部分 因特网算法	
7.1 单源点最短路径	231	第9章 文本处理	284
7.1.1 Dijkstra算法	231	9.1 串和模式匹配算法	284
7.1.2 Bellman-Ford最短路径算法	236	9.1.1 串操作	285
7.1.3 有向无环图中的最短路径	238	9.1.2 蛮力模式匹配	286
7.2 所有顶点对之间的最短路径	240	9.1.3 Boyer-Moore算法	287
7.2.1 动态规划最短路径算法	240	9.1.4 Knuth-Morris-Pratt算法	289
7.2.2 利用矩阵相乘计算最短路径	241	9.2 trie	292
7.3 最小生成树	244	9.2.1 标准trie	292
7.3.1 Kruskal算法	245	9.2.2 压缩trie	294
7.3.2 Prim-Jarnik算法	249	9.2.3 后缀trie	295
7.3.3 Baruvka算法	251	9.2.4 搜索引擎	298
7.3.4 MST算法比较	253	9.3 文本压缩	298
7.4 Java示例: Dijkstra算法	253	9.3.1 赫夫曼编码算法	299
7.5 习题	256	9.3.2 修正贪心法	300
基础题	256	9.4 文本相似性测试	301
创新题	256	9.4.1 最长公共子序列问题	301
程序设计	257	9.4.2 应用动态规划求解LCS问题	301
7.6 本章注记	258	9.5 习题	303
第8章 网络流和匹配	259	基础题	303
8.1 流和割	260	创新题	304
8.1.1 流网络	260	程序设计	305
8.1.2 割	261	9.6 本章注记	305
8.2 最大流	262	第10章 数论和密码学	306
8.2.1 剩余容量和增大路径	262	10.1 与数有关的基本算法	306
8.2.2 Ford-Fulkerson算法	264	10.1.1 基本数论的一些事实	307
8.2.3 Ford-Fulkerson算法分析	266	10.1.2 欧几里得GCD算法	308
8.2.4 Edmonds-Karp算法	267	10.1.3 模运算	310
8.3 最大二分匹配	269	10.1.4 模指数运算	313
8.4 最小代价流	270	10.1.5 模乘法逆元	315
8.4.1 增大回路	271	10.1.6 素性测试	316
8.4.2 连续最短路径	272	10.2 密码计算	320
8.4.3 修改权值	273	10.2.1 对称加密模式	321
8.5 Java示例: 最小代价流	276	10.2.2 公钥密码系统	322

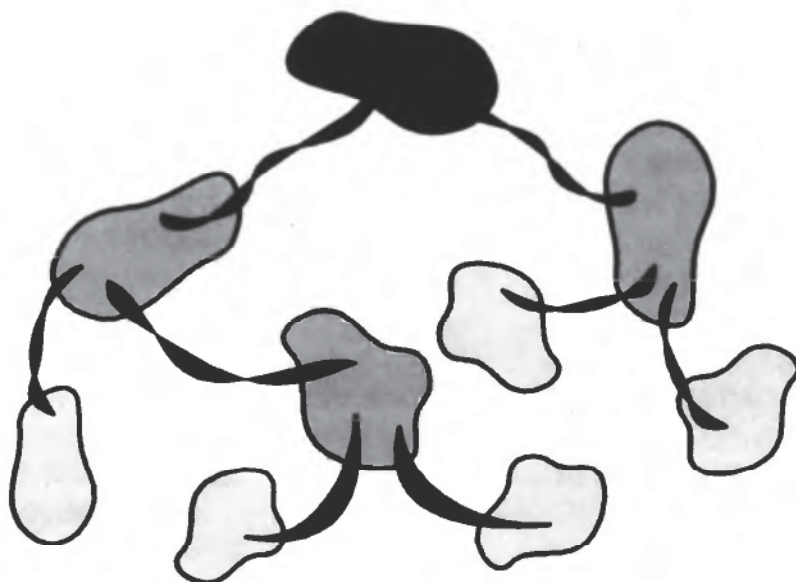
10.2.3	RSA密码系统	323	11.6	本章注记	369
10.2.4	El Gamal密码系统	325			
10.3	信息安全算法和协议	326		第四部分 其他主题	
10.3.1	单向散列函数	326		第12章 计算几何	372
10.3.2	时间戳和认证字典	327	12.1	范围树	372
10.3.3	硬币抛掷和比特承诺	327	12.1.1	一维范围查找	373
10.3.4	安全电子传输 (SET) 协议	328	12.1.2	二维范围查找	375
10.3.5	密钥分发和交换	329	12.2	优先查找树	377
10.4	快速傅里叶变换	330	12.2.1	构造一棵优先查找树	378
10.4.1	本原单位根	332	12.2.2	优先查找树中的查找	378
10.4.2	离散傅里叶变换	333	12.2.3	优先范围树	380
10.4.3	快速傅里叶变换算法	335	12.3	四叉树和 k -d树	380
10.4.4	大整数相乘	337	12.3.1	四叉树	381
10.5	Java示例: FFT	339	12.3.2	k -d树	382
10.6	习题	345	12.4	平面扫描技术	383
	基础题	345	12.4.1	正交线段相交	383
	创新题	345	12.4.2	查找最近点对	385
	程序设计	346	12.5	凸包	388
10.7	本章注记	346	12.5.1	几何对象表示	388
			12.5.2	点方位测试	389
第11章 网络算法		348	12.5.3	凸包的基本性质	390
11.1	复杂性测度和模型	349	12.5.4	礼品包扎算法	392
11.1.1	网络协议栈	349	12.5.5	Graham扫描算法	393
11.1.2	消息传递模型	349	12.6	Java示例: 凸包	395
11.1.3	网络算法的复杂性测度	350	12.7	习题	398
11.2	基本分布式算法	351		基础题	398
11.2.1	环网上的领导人选举	351		创新题	398
11.2.2	树网上的领导人选举	354		程序设计	400
11.2.3	广度优先查找	356	12.8	本章注记	400
11.2.4	最小生成树	359			
11.3	广播路由和单播路由	360		第13章 NP完全性	401
11.3.1	广播路由的洪泛算法	360	13.1	P类和NP类	402
11.3.2	单播路由的距离矢量算法	361	13.1.1	定义复杂类P和复杂类NP	402
11.3.3	单播路由的链路-状态算法	362	13.1.2	NP中的一些有趣问题	404
11.4	多播路由	363	13.2	NP完全性	406
11.4.1	逆向路径转发	363	13.2.1	多项式时间归约和NP困难度	406
11.4.2	中心树	364	13.2.2	Cook-Levin定理	407
11.4.3	Steiner树	365	13.3	重要的NP完全问题	408
11.5	习题	367	13.3.1	CNF-SAT和3SAT	410
	基础题	367	13.3.2	VERTEX-COVER	412
	创新题	367	13.3.3	CLIQUE和SET-COVER	413
	程序设计	369			

13.3.4	SUBSET-SUM和KNAPSACK	415	14.2.1	并行计算模型	445
13.3.5	HAMILTONIAN-CYCLE 和TSP	417	14.2.2	简单并行分治法	446
13.4	近似算法	419	14.2.3	串行子集和Brent定理	447
13.4.1	多项式时间的近似模式	420	14.2.4	递归倍增	447
13.4.2	VERTEX-COVER的2-近似 算法	422	14.2.5	并行归并和排序	450
13.4.3	TSP特例的2-近似算法	423	14.2.6	找出凸多边形的直径	450
13.4.4	SET-COVER的对数近似算法	424	14.3	在线算法	452
13.5	回溯法和分枝限界法	426	14.3.1	高速缓存算法	452
13.5.1	回溯法	426	14.3.2	拍卖策略	457
13.5.2	分枝限界法	429	14.3.3	竞争查找树	458
13.6	习题	433	14.4	习题	461
	基础题	433		基础题	461
	创新题	434		创新题	461
	程序设计	435		程序设计	462
13.7	本章注记	435	14.5	本章注记	462
第 14 章	算法框架	437	附录 A	有用的数学知识	464
14.1	外存算法	437	A.1	对数和指数	464
14.1.1	分层的存储器管理	438	A.2	整型函数和关系	465
14.1.2	(a, b) 树和B树	440	A.3	求和	466
14.1.3	外存排序	443	A.4	有用的数学技术	467
14.2	并行算法	445	参考书目	468	
			索引	476	

Part 1

第一部分

基础工具



第1章



算法分析

有一个经典的故事，国王要求著名数学家阿基米德确定工匠为其制作的皇冠是否是纯金的，而没有掺杂银。阿基米德在进入浴池的一霎那在浴池中找到了解决这个问题。他注意到水漫出浴池的量与他进入浴池的身体的体积成正比。明白这个事实蕴涵的道理后，他立即跑出浴池，裸奔穿过大街并大声喊道：“我找到了，我找到了！”因为他找到了一种分析工具（排水量），这种工具再加上一个简单的天平，就可以确定国王的新皇冠是否是纯金制成的。但是，这项发现对于金匠而言是不幸的，因为当阿基米德进行分析时，皇冠排出的水量比同等重量的纯金制品排出的水量要多，这就表明皇冠并非纯金制成。

本书着重研究设计“良好的”算法和数据结构。简言之，算法（algorithm）是一个在有限时间内逐步执行某项任务的过程，而数据结构（data structure）是一种系统地组织和访问数据的方法。这些都是计算的核心概念，但是为了能够确定什么是“良好的”算法和数据结构，就必须有精确的分析方法。

本书中所用的主要分析工具都涉及表征算法和数据结构操作的运行时间，以及占用空间的大小。因为时间是一种宝贵的资源，因而运行时间是“良好程度”的一种自然度量。但主要把运行时间作为良好程度的基本度量，意味着至少需要一点数学知识来描述运行时间和比较算法。

本章首先描述分析算法所需的基本框架，包括描述算法的语言、语言所适用的计算模型，以及计算运行时间时考虑的一些主要因素。还简明讨论了如何分析递归算法。在1.2节中，给出了用于表征运行时间的主要符号——所谓的“大O”符号。这些工具构成了设计和分析算法的主要理论工具。

在1.3节中，我们从对算法分析框架的讨论暂时转到对某些重要的数学知识的回顾，包括讨论求和、对数、证明技术和基本的概率知识。了解这个背景和算法分析所用的符号之后，1.4节提出了一些关于理论算法分析的案例研究。接下来在1.5节提出了称为平摊法的一项有趣的分析技术，它可以解释多个独立操作的共同动作。最后在1.6节讨论了一种重要和实用的分析技术——实验。我们讨论了一个良好的实验框架的主要原则，以及概括和表征实验分析数据的技术。

4

1.1 算法的分析方法学

算法或数据结构操作的运行时间通常与许多因素有关，那么什么才是度量它们的合理方法？如果算法已经实现，可以通过在各种测试输入下来执行它，并记录每次执行所使用的实际时间，

以研究它的运行时间。可以利用系统调用来精确地进行这种度量,这些系统调用是内置在算法的目标语言或操作系统中的。一般而言,只注重于确定运行时间与输入大小的相关性。为此可以用多种不同的输入大小,进行多次实验。然后可以用 x 坐标表示输入大小 n , y 坐标表示运行时间 t ,从而把算法每次运行的性能绘制成一个点,以使这些实验的结果可视化,如图1-1所示。为了使结果有意义,这种分析需要选择好的样本输入,并进行足够多的测试,这样才能够产生算法的合理统计描述。在1.6节会详细讨论这种方法。

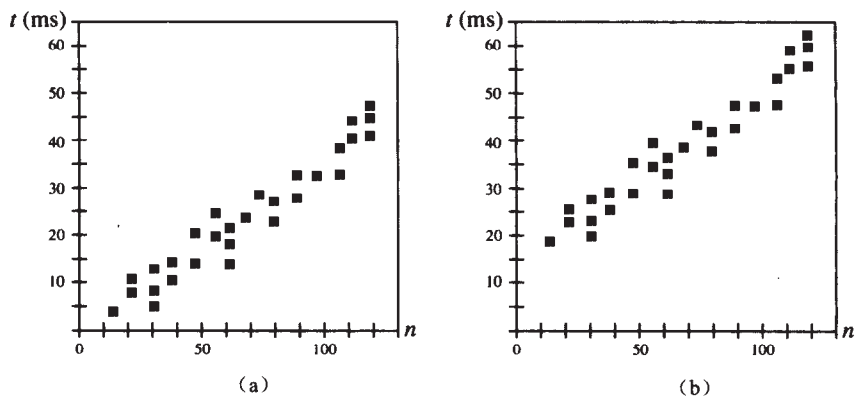


图1-1 对某算法运行时间的实验研究结果。坐标为 (n, t) 的点表示对于输入大小 n , 算法的运行时间为 t ms。(a) 在较快的计算机上执行的算法; (b) 在较慢的计算机上执行的算法

一般而言,算法或数据结构方法的运行时间随着输入大小的增加而增加,尽管也可能由于大小相同输入不同而有所变化。同时,算法的运行时间也受实现、编译和执行算法的硬件环境(处理器、时钟速率、内存、磁盘等)和软件环境(操作系统、程序设计语言、编译器、解释器等)的影响。在所有其他因素相同的情况下,如果计算机有快得多的处理器,或者实现算法的程序已编译成本机代码而不是运行在虚拟机上的解释实现,那么对于同一输入数据,同一算法的运行时间将会更短。

5

常规分析方法学的要求

对运行时间进行实验性研究是有用的,我们将在1.6节探讨这一点,但是它们有一些局限性:

- 实验只能在有限的测试输入集上进行,必须仔细考虑这些数据,以确保它们具有代表性。
- 除非在相同的硬件和软件环境上执行两个算法的运行时间的实验,否则难以比较两个算法的效率。
- 需要实现并执行一个算法,才能以实验方式研究它的运行时间。

因此,虽然实验在算法分析中起着重要的作用,但只用它进行算法分析是不够的。因此,除了进行实验外,还需要有一个分析框架,它可以:

- 考虑所有可能的输入。
- 允许独立于软、硬件环境来评估任意两个算法的相对效率。
- 无需真正实现算法和执行算法,通过研究算法的高级描述就能进行算法分析。

这种方法学的目标在于将每个算法与一个函数 $f(n)$ 关联起来,而 $f(n)$ 能够根据输入大小 n 表征算法的运行时间。典型的函数有 n 和 n^2 。例如,可以用这样的句子“算法 A 的运行时间与 n 成正比”表示如果要进行实验,会发现对于任意大小的输入 n ,算法 A 的实际运行时间永远不会超过 cn ,其中 c 为常数,与实验中所用的软、硬件环境有关。给定两个算法 A 和 B ,其中 A 的运行时间与 n 成正比, B 的运行时间与 n^2 成正比。算法 A 比算法 B 要好,因为函数 n 的增长速度比函数 n^2 的增长速

度要慢。

我们现在已经为开始研究算法分析方法学做好了准备。本方法学中包含如下几个要素：

- 描述算法的语言。
- 执行算法的目标计算模型。
- 度量算法运行时间的指标。
- 表征运行时间的方法，包括那些表征递归算法运行时间的方法。

6

本节的余下部分将详细描述这些要素。

1.1.1 伪代码

程序员常常被要求以一种常人能够读懂的方式描述算法。这样的描述不是计算机程序，但比通常的描述更结构化。它们还便于对数据结构和算法执行高级分析。这样的描述称为伪代码(pseudo-code)。

1. 伪代码实例

数组-最大值问题是指在存储 n 个整数的数组 A 中找出最大元素的简单问题。为了解决这个问题，可以利用名为arrayMax的算法，该算法利用一个for循环来遍历数组 A 中的元素。

算法arrayMax的伪代码描述如算法1-1所示。

算法1-1 算法arrayMax

```

算法 arrayMax( $A, n$ ):
  输入: 存储  $n \geq 1$  个整数的数组  $A$ 
  输出:  $A$  中的最大元素
   $currentMax \leftarrow A[0]$ 
  for  $i \leftarrow 1$  to  $n-1$  do
    if  $currentMax < A[i]$  then
       $currentMax \leftarrow A[i]$ 
  return  $currentMax$ 

```

注意，伪代码比等价的实际软件代码段更简洁。此外，伪代码更容易阅读和理解。

2. 利用伪代码证明算法正确性

通过检查伪代码，可以简略证明算法arrayMax的正确性。变量 $currentMax$ 开始时等于数组 A 中的第一个元素。在第 i 次循环迭代的开始，声明 $currentMax$ 等于 A 中前 i 个元素中的最大值。在第 i 次迭代中，将 $currentMax$ 与 $A[i]$ 进行比较，如果在本次迭代之前该声明为真，那么在 $i+1$ （计数器 i 的下一个值）次迭代以后它仍然为真。那么， $n-1$ 次迭代以后， $currentMax$ 将等于 A 中的最大元素。对于这个例子，我们希望伪代码的描述总是足够详尽，以便能够完全证明所描述算法的正确性，同时又足够简单，便于读者理解。

7

3. 什么是伪代码？

伪代码是自然语言和高级程序设计语言的混合物，它描述数据结构或算法的一般实现的主要思想。然而，由于伪代码语言依赖于自然语言，因而实际上伪代码语言没有精确的定义。同时，为了使叙述清楚，伪代码语言将自然语言和标准程序设计语言的结构结合起来。选择的程序设计语言构造与现代高级语言（如C、C++和Java）的那些构造一致，它们包含如下成分：

- **表达式：**利用标准数学符号表示数值表达式和布尔表达式。我们使用向左的箭头符号(\leftarrow)作为赋值语句中的赋值运算符（等价于C、C++和Java中的“=”运算符），使用等号(=)

表示布尔表达式中的相等关系（等价于C、C++和Java中的“==”关系）。

- 方法声明：算法 名（参数1，参数2，...）声明一个新的方法“名”及其参数。
- 判定结构：if 条件 then 执行条件为真时的动作 [else 执行条件为假时的动作]。我们使用缩进形式来指示条件为真或假时应该执行的动作。
- while循环：while 条件do动作。我们使用缩进形式来指示循环中应该包括的动作。
- repeat循环：repeat 动作 until 条件。利用缩进形式表明动作所属的循环体。
- for循环：for变量递增定义 do 动作。利用缩进形式表明动作所属的循环体。
- 数组下标： $A[i]$ 表示数组 A 中的第 i 个单元。包含 n 个单元的数组 A 中的单元下标为 $A[0] \sim A[n-1]$ ，与C、C++和Java保持一致。
- 方法调用：对象.方法（参数）（对象可选，如果它明了的话）。
- 方法返回值：return值。这个操作返回调用该方法的某个值。

编写伪代码时，切记是为读者而不是为计算机编写伪代码。因此，应该力图传达一些高级思想，而不是低级的实现细节。同时，对于重要步骤，不能一笔带过。就像人类的许多交流形式，找到合适的平衡是一项重要的技能，它可以通过实践逐步改进。

既然已经开发了描述算法的高级方式，下面讨论如何从分析角度表征用伪代码编写的算法。

8

1.1.2 随机存取机（RAM）模型

如上所述，实验性分析是有价值的，但却有其局限性。如果想要分析某个特定的算法，而又不执行针对该算法运行时间的实验，我们可以将下列更具分析性的方法直接应用于高级代码或伪代码。定义一个高级基本操作的集合，这些操作在很大程度上独立于所用的程序设计语言，并且能够在伪代码中确定。基本操作包括：

- 给变量赋值。
- 调用方法。
- 执行算术运算（例如，计算两个数的加法）。
- 比较两个数。
- 通过下标访问数组中的元素。
- 跟踪对象引用。
- 从方法中返回。

具体地讲，一个基本操作对应一条低级指令，它的执行时间与软、硬件环境有关，但不为常量。我们不是确定每个基本操作的某一特定执行时间，而只是统计有多少个基本操作要执行，并利用这个数 t 作为算法运行时间的一个高级估计值。这个操作计数与特定软、硬件环境中的实际运行时间有关，因为每个基本操作都对应于一条常数时间的指令，且基本操作的个数是固定的。这种方法中蕴涵的假设是不同基本操作的运行时间相当接近。因此，算法中执行的基本操作数 t 与那个算法的实际运行时间成正比。

RAM模型定义

只统计基本操作的方法所产生的计算模型称为随机存取机（random access machine, RAM）。不应当把这个模型和“随机存取存储器”相混淆，后者只是把计算机视作连接到许多内存单元的CPU。每个内存单元存储一个字，它可以是一个数，一个字符串，或一个地址，即基本类型的值。术语“随机存取”是指CPU利用一个基本操作随意访问内存单元的能力。为了保持模型简单，我们没有对内存的字中可以存储的数值的大小设置任何特定的限制。假设RAM模型中的CPU能够在恒定数量的步骤内执行任意基本操作，这些步骤与输入大小无关。因此，算法执行的基本操作

9 数的精确界限直接对应于RAM模型中该算法的运行时间。

1.1.3 统计基本操作的数量

现在以算法arrayMax为例，说明如何计算算法执行的基本操作数。算法1-1给出了它的伪代码。对于算法的每一步，统计它采取的基本操作数，并考虑到某些操作是重复执行的，因为这些操作包含在循环体内。

- 初始化变量currentMax为A[0]对应于两个基本操作（通过下标访问数组中的元素，并给变量赋值），并且在算法开始处仅执行一次。因此，基本操作计数为2个单位。
- 在for循环的开始处，将计数器i初始化为1。这个动作对应于执行一个基本操作（给变量赋值）。
- 在进入for循环体之前，验证条件*i* < *n*是否成立。这个动作对应于执行一个基本操作（比较两个数）。因为计数器*i*从0开始，并在循环的每次迭代之后增1，比较*i* < *n*将执行*n*次。因此，基本操作计数为*n*个单位。
- for循环体执行*n* - 1次（计数器取值为1, 2, ..., *n* - 1）。在每次迭代中，都会将A[*i*]与currentMax作比较（两个基本操作：通过下标访问数组元素和比较），可能将A[*i*]赋值给currentMax（两个基本操作：通过下标访问数组元素和赋值），计数器*i*递增（两个基本操作：求和与赋值）。因此，在循环的每次迭代中，要么执行4个基本操作，要么执行6个基本操作。这取决于A[*i*] ≤ currentMax或A[*i*] > currentMax。因此，循环体执行的基本操作数介于4(*n* - 1) ~ 6(*n* - 1)之间。
- 返回变量currentMax的值对应于一个基本操作，且只被执行一次。

总之，算法arrayMax执行的基本操作数*t*(*n*)至少为

$$2 + 1 + n + 4(n - 1) + 1 = 5n$$

至多为

$$2 + 1 + n + 6(n - 1) + 1 = 7n - 2$$

当A[0]为最大元素时，出现最好情况（*t*(*n*) = 5*n*），此时变量currentMax永不会被重新赋值。当元素按增序排列时，对于for循环的每次迭代，变量currentMax都会被重新赋值，从而出现最坏情况（*t*(*n*) = 7*n* - 2）。

10

平均情况分析和最坏情况分析

如同arrayMax方法那样，与另外一些输入相比，算法对于某些输入可能运行得更快一些。在这些情况下，希望将这类算法的运行时间表示为所有可能输入的一个平均值。尽管这样的平均情况分析通常是有价值的，但一般非常困难。它要求定义输入集上的概率分布，这通常是一项困难的任务。图1-2示意性地给出了与输入分布有关的一个算法的运行时间，它可以是介于最坏情况和最好情况之间的任意一种情况。例如，如果输入真的只是“A”或“D”，又会怎样呢？

平均情况分析通常要求根据给定的输入分布计算算法的期望运行时间。而这样的分析常常要求大量的数学和概率论知识。

因此，除了对本身随机化的算法进行实验性研究或分析之外，本书余下内容将根据最坏情况表征算法的运行时间。例如，假如算法arrayMax在最坏情况下执行*t*(*n*) = 7*n* - 2次基本操作。这表明算法所执行基本操作的最大数是所有输入大小*n*上的7*n* - 2。

因为这类分析不需要概率论的知识，它比平均情况分析简单得多。只要求能够确定最坏情况的输入，而这通常是直观的。此外，采用最坏情况的方法实际上可以导致更好的算法。制定一些

成功的标准,要使算法在最坏情况下执行良好,必须要求它对于每个输入都执行良好。也就是说,针对最坏情况设计可以获得更健壮算法,就像一位赛跑运动员总是通过跑步上山来进行训练。

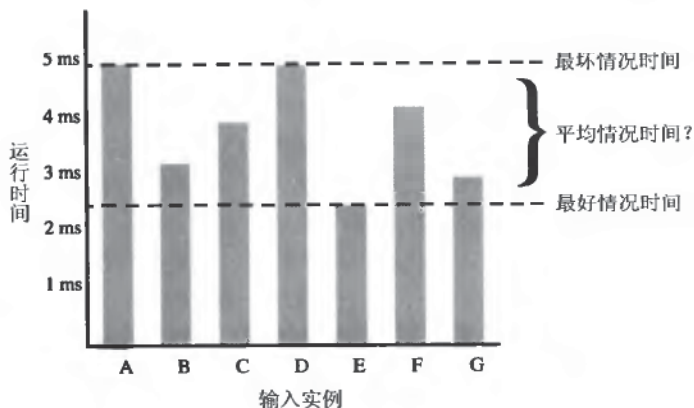


图1-2 最好情况和最坏情况的差别。每一柱形表示某个算法对于不同的可能输入的运行时间

11

1.1.4 递归算法分析

迭代法不是求解问题的唯一方法。递归 (recursion) 是许多算法使用的另一种方法。在递归法中,可以定义一个过程 P ,它能够将自己作为子例程来调用,这些对 P 的进一步调用能够用于求解更小的子问题。在更小实例上调用 P 的子例程称为“递归调用”。递归过程应当总是定义一个基准情况 (base case),它足够小,算法无需使用递归即可直接求解它。

算法1-2给出了求数组最大值问题的递归解。该算法首先检查数组中是否只含一个元素,如果是这种情况,则该元素必定为最大值;因此,在这种简单的基准情况中,可以立即求解问题。否则,算法将递归计算数组中前 $n-1$ 个元素的最大值,并返回该值和数组中最后一个元素之间的最大值。

算法1-2 算法recursiveMax

```

算法 recursiveMax( $A, n$ ):
    输入: 存储 $n(\geq 1)$ 个整数的数组 $A$ 
    输出:  $A$ 中的最大元素
    if  $n = 1$  then
        return  $A[0]$ 
    return max {recursiveMax( $A, n-1$ ),  $A[n-1]$ }
    
```

由此例可见,递归算法是相当精巧的。不过,分析递归算法的运行时间需要做一点额外的工作。确切地讲,要利用递归方程分析这种类型的运行时间。递归方程 (recurrence equation) 定义了递归算法的运行时间必须满足的数学公式。用函数 $T(n)$ 表示算法对于输入大小 n 的运行时间,并且编写 $T(n)$ 必须满足的方程。例如,可以把 recursiveMax 算法的运行时间 $T(n)$ 表征如下:

$$T(n) = \begin{cases} 3 & \text{如果 } n = 1 \\ T(n-1) + 7 & \text{其他情况} \end{cases}$$

假设将每个比较、数组引用、递归调用、最大值计算,或返回 (return) 看作一个基本操作。理想情况下,希望用封闭形式 (closed form) 表征类似上面的递归方程,即在方程右边不会出现对

函数 T 的引用。对于recursiveMax算法，容易看到的封闭形式为 $T(n) = 7(n-1) + 3 = 7n - 4$ 。一般而言，确定递归方程封闭形式的解比此例要困难得多。第4章在学习排序算法和选择算法时，会研究一些特殊递归方程的例子。5.2节将学习求解一般形式的递归方程的方法。

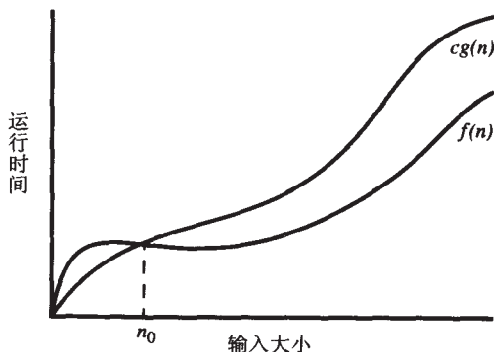
12

1.2 渐近符号

读者已经清楚评估一个简单的算法（如arrayMax）及其相关递归算法recursiveMax的运行时间的详细过程。如果要对更复杂的算法执行这种评估，显然，这样的方法是非常笨拙的。一般而言，伪代码描述中的每一步和高级语言实现中的每条语句都对应于少量的基本操作，它们不依赖于输入大小。因此，可以通过统计伪代码中的步骤数，或者执行的高级语言的语句数，来执行简化的分析，它会估计将执行的基本操作数，直到一个常数因子。幸运的是，有这么一种符号，可用于表征影响算法运行时间的主要因素，而无需透彻、准确地知道每个常数时间的指令集执行了多少个基本操作。

1.2.1 大O符号

设 $f(n)$ 和 $g(n)$ 是将非负整数映射为实数的函数。如果存在实常数 $c > 0$ 和整常数 $n_0 \geq 1$ ，对于每个 $n \geq n_0$ 的整数，满足 $f(n) \leq c \cdot g(n)$ ，则称 $f(n)$ 是 $O(g(n))$ 。这个定义常常称为大O符号，因为有时它读作“ $f(n)$ 是 $g(n)$ 的大O”。我们也可以将其读作“ $f(n)$ 是 $g(n)$ 阶的”。（图1-3说明了这个定义。）



13

图1-3 说明大O符号。当 $n \geq n_0$ 时，对于 $f(n) \leq c \cdot g(n)$ ，函数 $f(n)$ 是 $O(g(n))$

示例1.1 $7n-2$ 是 $O(n)$ 。

证明 由大O的定义可知，需要找一个实常数 $c > 0$ 和整常数 $n_0 \geq 1$ ，对于每个 $n \geq n_0$ 的整数，满足 $7n-2 \leq cn$ 。容易看出，一种可能的选择是 $c=7$ 和 $n_0=1$ 。实际上，这只是无数个可选方式之一，因为任何大于或等于7的实数 c ，以及任何大于或等于1的整数 n_0 都会满足条件。 ■

借助大O符号我们可以说，随着 n 趋向无穷大（依据定义中的语句“ $n \geq n_0$ ”），在渐近（asymptotic）意义上， n 的一个函数“小于或等于”另一个函数（依据定义中的不等号“ \leq ”）的常数倍（依据定义中的常数 c ）。

大O符号广泛用于表征以 n 为参数的运行时间和空间的界限， n 因问题而异，但通常定义为表示问题规模的直观概念。例如，如果要寻找整型数组中的最大元素（见算法1-1中给出的arrayMax算法），最自然的做法是，令 n 表示数组中的元素个数。例如，可以编写以下关于算法1-1中arrayMax算法的运行时间的精确陈述。

定理1.1 计算含 n 个整数的数组中最大元素的算法arrayMax的运行时间为 $O(n)$ 。

证明 如1.1.3节所示, 算法arrayMax中执行的基本操作数至多为 $7n-2$ 。利用大O的定义, 选取 $c=7$ 和 $n_0=1$, 可得算法arrayMax的运行时间为 $O(n)$ 。 ■

以下是说明大O符号的另外一些例子。

示例1.2 $20n^3 + 10n \log n + 5$ 是 $O(n^3)$ 。

证明 对于 $n \geq 1$, $20n^3 + 10n \log n + 5 \leq 35n^3$ 。 ■

实际上, 任何多项式 $a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$ 总是 $O(n^k)$ 。

示例1.3 $3 \log n + \log \log n$ 是 $O(\log n)$ 。

证明 对于 $n \geq 2$, $3 \log n + \log \log n \leq 4 \log n$ 。注意, 对于 $n=1$, $\log \log n$ 无定义, 故选 $n \geq 2$ 。 ■

示例1.4 2^{100} 是 $O(1)$ 。

证明 对于 $n \geq 1$, $2^{100} \leq 2^{100} \times 1$ 。注意, 变量 n 并没有出现在不等式中, 因为, 我们处理的是恒定值函数。 ■

示例1.5 $5/n$ 是 $O(1/n)$ 。

证明 对于 $n \geq 1$, $5/n \leq 5(1/n)$ (即使这实际上是一个递减函数)。 ■

14

一般而言, 应该用大O符号尽可能接近地表征一个函数。虽然函数 $f(n) = 4n^3 + 3n^{4/3}$ 为 $O(n^5)$ 并不错, 但更准确地说 $f(n)$ 是 $O(n^3)$ 。打个比方, 想像一位饥饿的旅游者驱车沿着一段很长的乡村公路行驶, 而一个当地的农民从市场步行回家的情景。如果旅游者问农民他要驾驶多长时间才能找到食物, 农民可能真诚地说“肯定不超过12小时”, 但是更准确 (并且更有帮助) 的回答应为“你只需在这条路上驾驶几分钟就能找到市场”。

通常并不总是直接利用大O定义得到问题的大O表征, 而可以利用以下规则简化表示。

定理1.2 设 $d(n)$ 、 $e(n)$ 、 $f(n)$ 和 $g(n)$ 是将非负整数映射到非负实数的函数。则

- (1) 如果 $d(n)$ 是 $O(f(n))$, 那么对于任何常数 $a > 0$, $ad(n)$ 是 $O(f(n))$ 。
- (2) 如果 $d(n)$ 是 $O(f(n))$, $e(n)$ 是 $O(g(n))$, 那么 $d(n) + e(n)$ 是 $O(f(n) + g(n))$ 。
- (3) 如果 $d(n)$ 是 $O(f(n))$, $e(n)$ 是 $O(g(n))$, 那么 $d(n) e(n)$ 是 $O(f(n) g(n))$ 。
- (4) 如果 $d(n)$ 是 $O(f(n))$, $f(n)$ 是 $O(g(n))$, 那么 $d(n)$ 是 $O(g(n))$ 。
- (5) 如果 $f(n)$ 是次数为 d 的多项式 (即, $f(n) = a_0 + a_1 n + \dots + a_d n^d$), 那么, $f(n)$ 是 $O(n^d)$ 。
- (6) 对于任意固定的 $x > 0$ 和 $a > 1$, n^x 是 $O(a^n)$ 。
- (7) 对于任意固定的 $x > 0$, $\log n^x$ 是 $O(\log n)$ 。
- (8) 对于任意固定的常数 $x > 0$ 和 $y > 0$, 则 $\log^x n$ 是 $O(n^y)$ 。

在大O符号中包含常数因子和低阶项被认为是不好的。例如, 尽管称函数 $2n^2$ 是 $O(4n^2 + 6n \log n)$ 是完全正确的, 但它是不合适的。我们应该试图用大O的最简单形式描述函数。

示例1.6 $2n^3 + 4n^2 \log n$ 是 $O(n^3)$ 。

证明 应用定理1.2中的规则, 如下:

- $\log n$ 是 $O(n)$ (规则8)。
- $4n^2 \log n$ 是 $O(4n^3)$ (规则3)。

- $2n^3 + 4n^2 \log n$ 是 $O(2n^3 + 4n^3)$ (规则2)。
- $2n^3 + 4n^3$ 是 $O(n^3)$ (规则5或规则1)。
- $2n^3 + 4n^2 \log n$ 是 $O(n^3)$ (规则4)。

■

在算法和数据结构的分析中经常出现某些函数，我们常用一些特殊术语称呼它们。表1-1给出了算法分析中的一些常用术语。

表1-1 表示函数分类的术语

对数	线性	二次	多项式	指数
$O(\log n)$	$O(n)$	$O(n^2)$	$O(n^k) (k \geq 1)$	$O(a^n) (a > 1)$

15

大O符号的使用

因为大O符号已经代表“小于或等于”概念，通常不写成“ $f(n) \leq O(g(n))$ ”。同样尽管“ $f(n) = O(g(n))$ ”这种说法很常见，但它并不完全正确（就通常理解的“=”关系而言），而且“ $f(n) \geq O(g(n))$ ”或“ $f(n) > O(g(n))$ ”实际上都不正确。最好说“ $f(n)$ 是 $O(g(n))$ ”。如果更偏向于用数学语言，称

$$f(n) \in O(g(n))$$

也是正确的。因为大O表示从技术上说定义了函数的全集。

即使有了这样的解释，在进行带有大O符号的算术运算时，也有相当的自由度，只要读者清楚运算与大O定义的联系。例如：称

$$f(n) \text{ 是 } g(n) + O(h(n))$$

意味着存在常数 $c > 0$ 和 $n_0 \geq 1$ ，对于 $n \geq n_0$ ，满足 $f(n) \leq g(n) + ch(n)$ 。如本例所示，有时只希望给出渐近表征中准确的最高项。在那种情况下，可以说“ $f(n)$ 是 $g(n) + O(h(n))$ ”，其中 $h(n)$ 的增长速度比 $g(n)$ 慢。例如，称 $2n \log n + 4n + 10\sqrt{n}$ 是 $2n \log n + O(n)$ 。

1.2.2 与大“O”相关的渐近符号

正如大O符号提供了称一个函数“小于或等于”另一个函数的渐近方式，还存在进行其他渐近方式比较的符号。

1. 大Ω和大Θ

设 $f(n)$ 和 $g(n)$ 是将整数映射为实数的函数。如果存在实常数 $c > 0$ 和整常数 $n_0 \geq 1$ ，对于 $n \geq n_0$ ，满足 $f(n) \geq cg(n)$ ，则称 $f(n)$ 是 $\Omega(g(n))$ （读作“ $f(n)$ 是 $g(n)$ 的大Ω”）。在渐近意义上，该符号的定义可用来表示一个函数大于或等于另一个函数的常数倍。同样，如果 $f(n)$ 是 $O(g(n))$ 且 $f(n)$ 是 $\Omega(g(n))$ ，则称 $f(n)$ 是 $\Theta(g(n))$ （读作“ $f(n)$ 是 $g(n)$ ”的大Θ）；即，存在实常数 $c' > 0$ 和 $c'' > 0$ ，以及整常数 $n_0 \geq 1$ ，对于 $n \geq n_0$ ，满足 $c'g(n) \leq f(n) \leq c''g(n)$ 。

大Θ符号可用于表示两个函数是渐近相等的，相差一个常数因子。下面将考察关于这些符号的一些例子。

16

示例1.7 $3 \log n + \log \log n$ 是 $\Omega(\log n)$ 。

证明 $3 \log n + \log \log n \geq 3 \log n$ ，对于 $n \geq 2$ 。

■

这个例子表明，在用大Ω符号建立下界时，低阶项可以忽略。因此，作为总结，下一个例子表明在Θ符号中，低阶项也可以忽略。

示例1.8 $3 \log n + \log \log n$ 是 $\Theta(\log n)$ 。

证明 由示例1.3和示例1.7可得。 ■

2. 一些需要注意的事项

在使用渐近符号时,有一些需要注意的事项。首先,注意到大O和相关符号的用法可能会被误用。它们“隐藏的”常数因子可能非常大。例如,虽然说函数 $10^{100}n$ 为 $\Theta(n)$ 是正确的,但如果这表示的是与运行时间为 $10n\log n$ 的算法作比较的算法的运行时间,则应该优先选择 $\Theta(n\log n)$ 时间算法,即使该问题的线性算法渐近地会变得更慢。这是因为常数因子 10^{100} 是一个“巨大的数”,许多天文学家认为该数字是可见宇宙中原子数的上界。因而,现实世界中不存在以此数作为输入的问题。因此,甚至在利用大O符号时,还是应该留心隐藏的常数因子和低阶项。

上述观察提出了一个“快速”算法由什么构成的问题。一般而言,运行时间为 $O(n\log n)$ (其常数因子大小合理)的算法应该认为是有效的。在某些情况下,当 n 较小时,甚至 $O(n^2)$ 时间方法也足够快。但一个 $O(2^n)$ 时间算法永远都不会被认为是有效的。这可通过关于国际象棋发明者的一则著名的故事进行说明。他要求国王在棋盘的第一个格子上放一粒稻米,在第二个格子上放2粒稻米,第三个格子上放4粒稻米,第四个格子上放8粒稻米,依此类推。试想将 2^{64} 粒稻米放在最后一个格子上的情景!事实上,在大多数程序设计语言中,甚至用标准的长整数也不能表示出这个数。

于是,如果要区分有效算法和效率低下的算法,可将它们划分成两类:那些运行时间为多项式时间的算法为一类,那些运行时间需要指数时间的算法为一类。也就是,对于某个常数 $k \geq 1$,运行时间为 $O(n^k)$ 的算法为一类,对于某些常数 $c > 1$,运行时间为 $\Theta(c^n)$ 的算法为一类。正像本节中讨论的许多概念一样,对于运行时间为 $\Theta(n^{100})$ 的算法,不可能被视为有效算法。即便如此,多项式时间算法和指数时间算法之间的区别仍被认为是问题难度的一个健壮的度量。

17

3. 与大O相关的其他符号:小o和小ω

还存在另一些符号,来表达一个函数严格小于或严格大于另一个函数。但这些符号不像大O、大Ω和大Θ那样常用。然而,出于完整性考虑,还是给出它们的定义。

设 $f(n)$ 和 $g(n)$ 是将整数映射为实数的函数。对于任意常数 $c > 0$,存在常数 $n_0 > 0$,对于 $n \geq n_0$,满足 $f(n) \leq cg(n)$,则称 $f(n)$ 是 $o(g(n))$ (读作“ $f(n)$ 是 $g(n)$ 的小o”)。若对于任意常数 $c > 0$,存在常数 $n_0 > 0$,对于 $n \geq n_0$,满足 $cg(n) \leq f(n)$,则称 $f(n)$ 是 $\omega(g(n))$ (读作“ $f(n)$ 是 $g(n)$ 的小ω”)。直观上讲, $o(\cdot)$ 在渐近意义上类似“小于”, $\omega(\cdot)$ 在渐近意义上类似“大于”。

示例1.9 函数 $f(n) = 12n^2 + 6n$ 是 $o(n^3)$ 和 $\omega(n)$ 。

证明 首先证明 $f(n)$ 是 $o(n^3)$ 。设 $c(>0)$ 为常数。如果取 $n_0 = (12+6)/c$,那么,对于 $n \geq n_0$,可得

$$cn^3 \geq 12n^2 + 6n^2 \geq 12n^2 + 6n$$

因此, $f(n)$ 是 $o(n^3)$ 。

为了证明 $f(n)$ 是 $\omega(n)$ 。再次设 $c(>0)$ 为常数。如果取 $n_0 = c/12$,那么,对于 $n \geq n_0$,可得

$$12n^2 + 6n \geq 12n^2 \geq cn$$

因此, $f(n)$ 是 $\omega(n)$ 。 ■

对于那些熟悉极限的读者,假设极限存在, $f(n)$ 是 $o(g(n))$,当且仅当

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

小 o 符号和大 O 符号之间的主要区别在于： $f(n)$ 是 $O(g(n))$ 表示如果存在常数 $c > 0$ 和 $n_0 \geq 1$ ，对于 $n \geq n_0$ ，满足 $f(n) \leq cg(n)$ ；而 $f(n)$ 是 $o(g(n))$ 表示对于所有常数 $c > 0$ ，存在常数 $n_0 > 0$ ，对于 $n \geq n_0$ ，满足 $f(n) \leq cg(n)$ 。直观上讲， $f(n)$ 是 $o(g(n))$ ，随着 n 趋向无穷大，与 $g(n)$ 相比， $f(n)$ 是不重要的。如前所述，渐近符号的作用使得我们可以只关注决定函数增长的主要因素。

总之，渐近符号大 O 、大 Ω 、大 Θ ，以及小 o 和小 ω 为我们提供了一种分析数据结构和算法的方便语言。如前所述，有了这些符号非常方便，因为这使我们可以只关注主要因素，而忽略一些低阶细节。

18

1.2.3 渐近表示的重要性

渐近符号有许多重要的好处，也可能这些好处不是那么明显。表1-2描述了渐近表示的一个重要方面。表中列出了不同算法对于某个输入实例在1秒、1分和1小时内能求解的最大问题规模，假定在1微秒(μs)内可以处理一个操作。该表同时表明算法设计的重要性。即使较快算法所含的常数因子较大，从长远观点来看，一个渐近运行时间较快（例如运行时间为 $O(n \log n)$ ）的算法优于一个渐近运行时间较慢（例如运行时间为 $O(n^2)$ ）的算法。

表1-2 1秒、1分和1小时内能求解的最大问题规模，不同的运行时间以微秒度量

运行时间	最大问题规模(n)		
	1 秒	1 分	1 小时
$400n$	2 500	150 000	9 000 000
$20n \lceil \log n \rceil$	4 096	166 666	7 826 087
$2n^2$	707	5 477	42 426
n^4	31	88	244
2^n	19	25	31

设计一个良好的算法要比在一台给定的计算机上有效解决问题重要。然而，正如在表1-3中所显示的那样，即使实现显著的硬件加速，仍然不能克服一个渐近较慢算法所引起的障碍。表中显示了对于任何固定运行时间可达到的新的最大问题规模，假定具有给定运行时间的算法现在运行在比前一台计算机快256倍的计算机上。

表1-3 在某一固定时间内，利用一台比原来快256倍的计算机，对于具有各种不同运行时间的算法，所能求解的最大问题规模将会增加。 m 是原最大问题规模，每一项都是 m 的函数

运行时间	新的最大问题规模
$400n$	$256m$
$20n \lceil \log n \rceil$	近似 $256((\log m)/(7 + \log m))m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m + 8$

19

根据函数增长速率排列函数

假定有两个算法可用于求解同一个问题：算法 A 的运行时间为 $\Theta(n)$ ，算法 B 的运行时间为 $\Theta(n^2)$ 。哪一个算法更好呢？小 o 符号称 n 是 $o(n^2)$ ，这意味着算法 A 渐近优于（asymptotically better）算法 B 。尽管对于较小的 n ，算法 B 的运行时间可能比算法 A 短。然而在长时间的运行中，如上面几个表所示，显然算法 A 优于算法 B 。

一般而言,可以利用小o符号,根据渐近增长速率,排列函数的类别。在表1-4中,列出了按照增长速率排列的函数,即如果表中的一个函数 $f(n)$ 排在另一个函数 $g(n)$ 之前,那么 $f(n)$ 是 $o(g(n))$ 。

表1-4 简单函数的排序列表。注意,使用常用术语,表中包括一个对数函数、两个对数多项式函数、三个亚线性函数、一个线性函数、一个二次函数、一个三次函数和一个指数函数

按增长速率排列的函数	
$\log n$	
$\log^2 n$	
\sqrt{n}	
n	
$n \log n$	
n^2	
n^3	
2^n	

表1-5中说明了表1-4中的函数在增长速率上的区别,除了函数 $\log^2 n$ 之外。

表1-5 多个函数的增长速率

n	$\log n$	\sqrt{n}	n	$n \log n$	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	2.8	8	24	64	512	256
16	4	4	16	64	256	4 096	65 536
32	5	5.7	32	160	1 024	32 768	4 294 967 296
64	6	8	64	384	4 096	262 144	1.84×10^{19}
128	7	11	128	896	16 384	2 097 152	3.40×10^{38}
256	8	16	256	2 048	65 536	16 777 216	1.15×10^{77}
512	9	23	512	4 608	262 144	134 217 728	1.34×10^{134}
1 024	10	32	1 024	10 240	1 048 576	1 073 741 824	1.79×10^{308}

20

1.3 数学概览

本节简要回顾离散数学的一些基本概念,这些概念将会出现在我们的多个讨论中。除了这些基本概念外,附录A还列出了数据结构和算法分析中使用的其他有用的数学知识。

1.3.1 求和

数据结构和算法分析中一再遇到的一种符号就是求和 (summation), 其定义如下

$$\sum_{i=a}^b f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b)$$

在数据结构和算法分析中会发生求和,这是由于循环的运行时间自然会导致求和。例如,数据结构和算法分析中通常发生的是几何求和。

定理1.3 对于任何 ≥ 0 的整数 n , 和任何 >0 且 $\neq 1$ 的实数 a , 考虑

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \cdots + a^n$$

(记住如果 $a > 0$, 则 $a^0 = 1$)。这个求和等于

$$\frac{1-a^{n+1}}{1-a}$$

定理1.3中所示的求和称为几何 (geometric) 求和。因为如果 $a > 1$, 则每一项都比前一项呈几何级数增大。也就是说, 这个几何求和中的项呈指数增长。例如, 从事计算的人都知道

$$1 + 2 + 4 + 8 + \dots + 2^{n-1} = 2^n - 1$$

因为这是 n 位二进制表示法中所能表示的最大整数。

在很多环境中发生的另一种的求和是

$$\sum_{i=1}^n i = 1 + 2 + 3 + \dots + (n-2) + (n-1) + n$$

这种求和经常出现在对循环所进行的分析中。在这种情况下, 对于循环的每次迭代, 在循环内部执行的操作的数量随着每次迭代都会增加一个固定的常数。这个求和还有一段有趣的历史。1787年, 一个德国小学教师决定让他的9年级和10年级学生执行一项从1加到100的加法任务。但就在给出这份作业之后, 其中一个孩子马上给出了答案, 即5050。

这个小学生正是卡尔·高斯 (Karl Gauss), 他后来成长为19世纪最伟大的数学家之一。人们普遍认为年轻的高斯是用以下等式解答了老师的作业。

定理1.4 对于任何 $n \geq 1$ 的整数, 有

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

证明 图1-4给出了定理1.4的两种可视化证明。这两种证明都是基于计算表示数 $1 \sim n$ 的矩形集合的面积。在图1-4a中, 在一系列矩形上绘制一个大三角形, 注意矩形的面积等于大三角形的面积 ($n^2/2$) 加上 n 个小三角形的面积, 每个小三角形的面积为 $1/2$ 。在图1-4b中, 当 n 为偶数时可应用这种证明方法, 注意 $1 + n = n + 1$, $2 + n - 1 = n + 1$, $3 + n - 2 = n + 1$, 依此类推。共有 $n/2$ 个这样的数对。

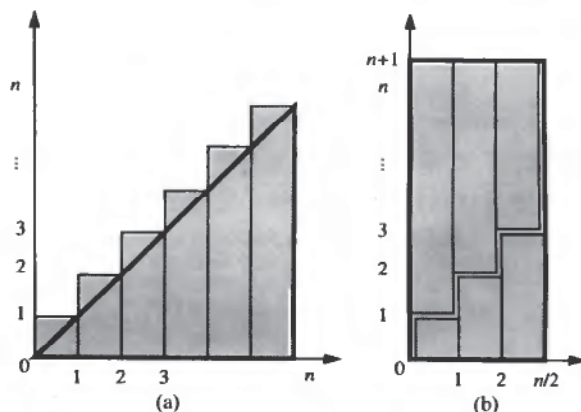


图1-4 定理1.4的可视化证明。两种方式说明了同一个问题, 即计算由宽 n 个单位、高分别为 $1, 2, \dots, n$ 的矩形覆盖的整个区域的面积。在图1-4a中, 显示的矩形包含一个面积为 $n^2/2$ 的大三角形 (底为 n , 高为 n) 和 n 个面积为 $1/2$ 的小三角形 (底为 1 , 高为 1) 构成。在图1-4b中, 这种方法仅应用于 n 为偶数时, 显示的矩形包含一个底为 $n/2$ 、高为 $n+1$ 的大矩形

1.3.2 对数和指数

数据结构和算法分析中有趣、有时甚至令人惊讶的事情之一是对数和指数无处不在。

如果 $a = b^c$, 则 $\log_b a = c$

如计算机科学文献中约定的那样, 当 $b = 2$ 时, 省略对数的底数 b 。例如, $\log 1024 = 10$ 。

关于对数和指数有许多重要法则。以下定理列出了其中一些:

定理1.5 设 a 、 b 和 c 是正实数, 则有:

- (1) $\log_b ac = \log_b a + \log_b c$
- (2) $\log_b a/c = \log_b a - \log_b c$
- (3) $\log_b a^c = c \log_b a$
- (4) $\log_b a = (\log_c a) / \log_c b$
- (5) $b^{\log_c a} = a^{\log_c b}$
- (6) $(b^a)^c = b^{ac}$
- (7) $b^a b^c = b^{a+c}$
- (8) $b^a / b^c = b^{a-c}$

同时, 作为简略符号, 利用 $\log^c n$ 表示函数 $(\log n)^c$, 利用 $\log \log n$ 表示 $\log (\log n)$ 。我们没有给出上述等式的推导过程, 它们全都遵循对数及指数的定义。以下利用几个例子说明这些等式的用法。

示例1.10 演示一些当对数或指数的基数为2时的有趣情况。引用的规则参考定理1.5。

- $\log (2n \log n) = 1 + \log n + \log \log n$, 根据规则1(应用两次)
- $\log (n/2) = \log n - \log 2 = \log n - 1$, 根据规则2
- $\log (\sqrt{n}) = \log (n)^{1/2} = (\log n)/2$, 根据规则3
- $\log \log \sqrt{n} = \log (\log n)/2 = \log \log n - 1$, 根据规则2和3
- $\log_4 n = (\log n) / \log 4 = (\log n) / 2$, 根据规则4
- $\log 2^n = n$, 根据规则3
- $2^{\log n} = n$, 根据规则5
- $2^{2 \log n} = (2^{\log n})^2 = n^2$, 根据规则5和6
- $4^n = (2^2)^n = 2^{2n}$, 根据规则6
- $n^2 2^{3 \log n} = n^2 \times n^3 = n^5$, 根据规则5、6和7
- $4^n / 2^n = 2^{2n} / 2^n = 2^{2n-n} = 2^n$, 根据规则6和8

23

向下取整函数和向上取整函数

另一个与对数有关的问题是取整。一个对数的值通常不为整数, 然而, 算法的运行时间一般用整数量表示, 如执行的操作个数。因此, 算法分析有时涉及使用所谓的“向下取整”或“向上取整”函数。分别定义如下:

- $\lfloor x \rfloor$ = 小于或等于 x 的最大整数。
- $\lceil x \rceil$ = 大于或等于 x 的最小整数。

这些函数给出了将一个实值函数转换成整数值函数的方式。即便如此, 用于分析数据结构和算法的函数经常被简单地表示成实值函数(例如, $n \log n$ 或 $n^{3/2}$)。我们应该将这样一个运行时间读作包围它的“大”向上取整函数。¹

1. 实值运行时间函数几乎总是与1.2节描述的渐近符号一起使用, 因为不管怎样, 使用向上取整函数通常是多余的(见习题R-1.24)。

1.3.3 简单证明技术

有时希望对某种数据结构或算法进行论证。例如，希望证明算法是正确的，或者它的运行速度快。为了严格进行此类声明，必须利用数学语言，并且为了支持这些声明，还要论证或证明我们的声明。幸运的是，有几种简单的方式可以做到这一点。

1. 通过实例

有些声明具有一般形式。“集合 S 中存在一个元素 x ，具有性质 P ”。为证明这条声明，只需找出某个 $x \in S$ ，且具有性质 P 。同样，某些难以确认的声明也具有一般形式，“集合 S 中的每个元素 x 都具有性质 P ”。为了证明这个声明为假，只需找出 S 中的某个 x 不具有性质 P 。这样的实例称为反例（counterexample）。

示例1.11 某个教授Amongus声明每个形如 $2^i - 1$ 的数为素数，其中 i 为大于1的整数。Amongus教授是错误的。

证明 为了证明Amongus教授的声明是错误的，需要找出一个反例。幸运的是，不需看的太远，因为 $2^4 - 1 = 15 = 3 \times 5$ 。 ■

24

2. “反面”进攻

另一组证明技术涉及否定的使用。有两种基本方法使用了逆否命题法（contrapositive）和反证法（contradiction）。逆否命题法就像看反面镜子。为了证明“如果 p 为真，那么 q 为真”的声明，建立“如果 q 不为真，那么 p 也不为真”的声明。逻辑上，这两个声明是相同的，但后者（称为前者的逆否命题）更容易想明白。

示例1.12 如果 ab 为奇数，那么 a 是奇数，或 b 是偶数。

证明 为了证明这个声明，考虑它的逆否命题，“如果 a 是偶数，且 b 是奇数，那么 ab 为偶数。”因此，假定对于某个整数 i ， $a = 2i$ ，那么 $ab = (2i)b = 2(ib)$ ；因此， ab 为偶数。 ■

除了显示逆否命题技术的使用之外，上一个例子还包括利用DeMorgan定律。这个定律可以帮助我们处理否命题。例如，形如“ p 或 q （ $p \vee q$ ）”的声明的否命题为“非 p 且非 q （ $\neg p \wedge \neg q$ ）”。同样，形如“ p 与 q （ $p \wedge q$ ）”的声明的否命题为“非 p 或非 q （ $\neg p \vee \neg q$ ）”。

另一种否定证明技术是反证法，它常常涉及利用DeMorgan定律。在利用反证法进行证明的过程中，首先建立一个真命题 q ，假定 q 为假，然后证明这个假设导致矛盾（ $2 \neq 2$ 或 $1 > 3$ ）。通过这个矛盾，证明不存在使 q 为假的情况，于是 q 一定为真。当然为了得到这个结论，必须确信假设 q 为假之前，所有情况一致。

示例1.13 如果 ab 为奇数，那么 a 是奇数，或 b 是偶数。

证明 设 ab 为奇数。要证明 a 是奇数，或 b 是偶数。因此希望导出矛盾，可以做出相反的假设，即假设 a 是偶数，或 b 是奇数，则对于某个整数 i ， $a = 2i$ 。因此， $ab = (2i)b = 2(ib)$ ，即 ab 为偶数，这与 ab 为奇数的条件矛盾， ab 不可能同时又为奇数又为偶数。因此， a 是奇数，或 b 是偶数。 ■

3. 归纳法

我们所做的大多数关于运行时间和空间界限的声明都与一个整型参数 n （通常直观地表示问题的规模）有关。此外，这些声明中的大多数都等价于说“对于所有的 $n \geq 1$ ”，某些声明 $q(n)$ 为真。由于这是对一个无限的数字集合所做的声明，不能直接穷举证明它。

不过，我们经常利用归纳法（induction）证明上述声明为真。这项技术表明，对于任何特定

的 $n \geq 1$, 存在一个有限蕴涵序列, 它从一个已知为真的序列开始, 最终推导出 $q(n)$ 为真。从 $n = 1$ (也可能是其他值 $n = 2, 3, \dots, k$, k 为常数) 开始证明 $q(n)$ 为真, 然后证明归纳“步骤”对于 $n > k$ 为真, 即证明“如果 $q(i)$ 对于 $i < n$ 为真, 那么 $q(n)$ 为真”。组合这两步即完成归纳法的证明。

25

示例1.14 考虑斐波那契序列: $F(1) = 1, F(2) = 2$, 以及对于 $n \geq 2$, 有 $F(n) = F(n-1) + F(n-2)$ 。声明 $F(n) < 2^n$ 。

证明 用归纳法证明该声明是正确的。

归纳基础 (base case): ($n \leq 2$)。 $F(1) = 1 < 2 = 2^1$, $F(2) = 2 < 4 = 2^2$ 。

归纳步骤 (induction step): ($n > 2$)。假定声明对于 $n' < n$ 为真。考虑 $F(n)$ 。因为 $n > 2$, $F(n) = F(n-1) + F(n-2)$ 。此外, 由于 $n-1 < n$, $n-2 < n$, 利用归纳假设 (inductive assumption, 有时称为 inductive hypothesis) 可得 $f(n) < 2^{n-1} + 2^{n-2}$ 。此外

$$2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2 \times 2^{n-1} = 2^n$$

定理证毕。

做另一个归纳证明技术的例子, 这个例子此前出现过。

定理1.6 (同定理1.4)

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

证明 用归纳法证明该等式。

归纳基础: $n = 1$ 。如果 $n = 1$, 由于 $1 = n(n+1)/2$, 定理平凡成立。

归纳步骤: $n \geq 2$ 。假定声明对于 $n' < n$ 为真。考虑 n 。

$$\sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i$$

由归纳假设, 可得

$$\sum_{i=1}^n i = n + \frac{(n-1)n}{2}$$

化简得

$$n + \frac{(n-1)n}{2} = \frac{2n + n^2 - n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2}$$

定理证毕。

有时感觉到要证明对于所有的 $n \geq 1$ 为真是一件困难的工作。但应该记住归纳技术的正确性。它表明对于某个特殊的 n , 存在一个有限的逐步蕴涵序列, 从某个为真的声明开始, 最终导出关于 n 的声明为真。简言之, 归纳证明是一个法则, 用于建立一系列的直接证明。

26

4. 循环不变式

本节最后讨论的一个证明技术是循环不变式 (loop invariant)。

为证明某个声明 S 关于一个循环是正确的, 根据一系列更小的声明 S_0, S_1, \dots, S_k 定义 S , 其中:

(1) 初始声明 S_0 在循环开始之前为真。

(2) 如果声明 S_{i-1} 在第 i 次迭代开始之前为真, 那么可以证明声明 S_i 在第 i 次迭代结束之后也为真。

(3) 最终声明 S_k 蕴涵着要证明的声明 S 为真。

在1.11节 (证明arrayMax的正确性), 实际上已经看见过循环不变式证明技术。这里仍然给

出一些例子。特别地，考虑应用循环不变式方法，证明算法1-3所示的算法arrayFind的正确性。该算法在数组 A 中查找元素 x 。

算法1-3 算法arrayFind

```

算法 arrayFind( $x, A$ ):
    输入: 元素 $x$ 和具有 $n$ 个元素的数组 $A$ 
    输出: 满足 $x = A[i]$ 的下标, 或-1 (如果 $A$ 中不存在等于 $x$ 的元素)
     $i \leftarrow 0$ 
    while  $i < n$  do
        if  $x = A[i]$  then
            return  $i$ 
        else
             $i \leftarrow i + 1$ 
    return -1

```

为了证明arrayFind是正确的，利用循环不变式技术证明。也就是说，对于 $i = 0, 1, \dots, n$ ，归纳性地定义一些声明 S_i ，然后推导出arrayFind的正确性。在第 i 次迭代开始时，以下声明为真：

S_i : x 不等于 A 中前 i 个元素中的任意一个

在循环的第一次迭代开始时，声明为真。因为此时 A 中前0个元素不含元素（这种情况平凡成立，称为空成立）。在第 i 次迭代中，将元素 x 与元素 $A[i]$ 比较，如果它们相等，则返回下标 i ，这显然是正确的。如果这两个元素不相等，那么我们就发现有一个以上的元素不等于 x ，并使下标 i 加1。因此，在下一迭代开始时，对于这个新下标 i ，声明 S_i 为真。如果while循环终止时，没有返回 A 中任何下标，那么 S_n 为真——这说明 A 中不存在等于 x 的元素。因此，算法是正确的，并根据需要返回设定的非下标值-1。

27

1.3.4 概率基础

在分析使用随机化方法的算法时，或者想要分析一个算法的平均情况的性能时，都会用到一些基本的概率论知识。最基本的知识是关于概率的任何声明都定义在样本空间（sample space） S 的基础上， S 被定义为某个实验中所有可能的结果集合。然而，这里对于术语“结果”和“实验”未作形式定义。

示例1.15 考虑一个实验，它由投掷一枚硬币5次的结果组成。这个样本空间有 2^5 种不同的结果。每一种结果表示可能的投掷可以产生的每种不同的次序。

样本空间可以是无限的，以下例子说明了这一点。

示例1.16 考虑一个实验。在实验中，不断投掷一枚硬币，直到出现正面为止。这个样本空间是无限的，每个结果都是先出现一系列 i 个反面，其后接着一个出现正面的投掷， $i \in \{0, 1, 2, 3, \dots\}$ 。

概率空间（probability space）是由样本空间 S 和一个概率函数 Pr 组成的。 Pr 将 S 的某个子集映射为 $[0, 1]$ 区间中的实数。数学上它表示的是某个事件发生概率的概念。形式上，将 S 的每个子集 A 称为一个事件（event），假设概率函数 Pr 关于定义在 S 上的事件具有如下基本性质：

(1) $\text{Pr}(\emptyset) = 0$ 。

(2) $\text{Pr}(S) = 1$ 。

(3) 对于任何 $A \subseteq S$, $0 \leq \Pr(A) \leq 1$ 。

(4) 如果 $A, B \subseteq S$ 且 $A \cap B = \emptyset$, 那么 $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ 。

1. 独立性

如果对于事件 A 和 B , 有

$$\Pr(A \cap B) = \Pr(A) \times \Pr(B)$$

则称两个事件 A 和 B 是独立的 (independent)。

如果对于任何子集 $\{A_{i1}, A_{i2}, \dots, A_{ik}\}$, 有

$$\Pr(A_{i1} \cap A_{i2} \cap \dots \cap A_{ik}) = \Pr(A_{i1}) \Pr(A_{i2}) \cdots \Pr(A_{ik})$$

则称事件集 $\{A_1, A_2, \dots, A_n\}$ 是相互独立的 (mutually independent)。

示例1.17 设 A 表示掷一粒骰子得到点数为6的事件, B 表示掷另一粒骰子得到点数为3的事件, C 表示投掷这两粒骰子得到点数之和为10的事件。那么 A 和 B 是独立事件, 但 C 不独立于 A 或 B 。

28

2. 条件概率

给定事件 B , 事件 A 发生的条件概率 (conditional probability) 用 $\Pr(A|B)$ 表示, 定义为

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)}$$

假设 $\Pr(B) > 0$ 。

示例1.18 设 A 表示投掷两粒骰子得到点数之和为10的事件, B 表示投掷第一粒骰子得到点数为6的事件。注意 $\Pr(B) = 1/6$ 且 $\Pr(A \cap B) = 1/36$ 。因为如果第一粒骰子的点数为6, 则只有一种情况使得两粒骰子的点数之和为10 (即第二粒骰子的点数为4)。因此, $\Pr(A|B) = (1/36)/(1/6) = 1/6$ 。

3. 随机变量和期望

用随机变量 (random variable) 处理事件是一种简便的方式。直观上讲, 随机变量就是一些变量, 其值依赖于某个实验的结果。形式上, 随机变量是一个函数 X , 它将某个样本空间 S 的结果映射到实数上。指示器随机变量 (indicator random variable) 是将某些结果映射到集合 $\{0, 1\}$ 上的随机变量。在算法分析中, 常常利用具有可能结果离散集合的随机变量 X 表征随机化算法的运行时间。在这种情况下, 算法中使用的随机来源的所有可能结果定义了样本空间 S 。通常最感兴趣的是这种随机变量的典型值、平均值或期望值。离散型随机变量 X 的期望值 (expected value) 定义如下:

$$E(X) = \sum_x x \Pr(X = x)$$

其中求和定义在 X 域上。

定理1.7 (期望的线性度) 设 X 和 Y 是任意两个随机变量, 则 $E(X + Y) = E(X) + E(Y)$ 。

$$\begin{aligned} \text{证明 } E(X + Y) &= \sum_x \sum_y (x + y) \Pr(X = x \cap Y = y) \\ &= \sum_x \sum_y x \Pr(X = x \cap Y = y) + \sum_x \sum_y y \Pr(X = x \cap Y = y) \\ &= \sum_x \sum_y x \Pr(X = x \cap Y = y) + \sum_y \sum_x y \Pr(Y = y \cap X = x) \\ &= \sum_x x \Pr(X = x) + \sum_y y \Pr(Y = y) \\ &= E(X) + E(Y) \end{aligned}$$

29 注意当 X 和 Y 各自取值时, 这个证明不依赖于任何关于事件的独立性假设。 ■

示例1.19 设 X 是随机变量, 它将两粒均匀骰子的投掷结果设为所示点数之和。那么 $E(X) = 7$ 。

证明 为证明这个声明, 设 X_1 和 X_2 分别为对应于每粒骰子的点数的随机变量。因此, $X_1 = X_2$ (即它们是同一函数的两个实例), 且 $E(X) = E(X_1 + X_2) = E(X_1) + E(X_2)$ 。均匀骰子的每个投掷结果均以概率 $1/6$ 发生。因此, 对于 $i = 1, 2$,

$$E(X_i) = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = \frac{7}{2}$$

于是, $E(X) = 7$ 。 ■

如果对于所有的实数 x 和 y , 有

$$\Pr(X = x \mid Y = y) = \Pr(X = x)$$

则两个随机变量 X 和 Y 是独立的 (independent)。

定理1.8 如果两个随机变量 X 和 Y 是独立的, 那么

$$E(XY) = E(X)E(Y)$$

示例1.20 设 X 是随机变量, 它将两粒均匀骰子的投掷结果设为所示点数之积, 那么, $E(X) = 49/4$ 。

证明 设 X_1 和 X_2 分别为对应于每粒骰子点数的随机变量。那么, 显然 X_1 和 X_2 是独立的。因此,

$$E(X) = E(X_1 X_2) = E(X_1) E(X_2) = (7/2)^2 = 49/4$$

4. Chernoff界

在分析随机化算法时, 常常需要确定一组随机变量之和的界。可以求解这个问题的一个不等式集合是Chernoff界的集合。设 X_1, X_2, \dots, X_n 是一组相互独立的指示器随机变量, 其中每个 X_i 均为1的概率为 $p_i > 0$, 其他情况下每个 X_i 均为0。设 $X = \sum_{i=1}^n X_i$ 是这些随机变量的和, 设 μ 表示 X 的均值, 即 $\mu = E(X) = \sum_{i=1}^n p_i$ 。以下仅给出定理, 而没有给出证明。

定理1.9 设 X 如上定义。那么, 对于 $\delta > 0$,

$$\Pr(X > (1 + \delta)\mu) < \left[\frac{e^\delta}{(1 + \delta)^{(1 + \delta)}} \right]^\mu$$

且对于 $0 < \delta \leq 1$,

$$\Pr(X < (1 - \delta)\mu) < e^{-\mu\delta^2/2}$$

30

1.4 算法分析案例研究

在介绍了描述和分析算法的一般框架之后, 现在考虑一些算法分析案例研究。确切地讲, 将展示如何使用大O符号分析两个求解同一个问题但具有不同运行时间的算法。

本节关注所谓的计算一系列数的前缀平均值 (prefix average) 问题, 即给定存储 n 个数的数组 X , 要计算数组 A , 其中 $A[i]$ 为元素 $X[0], \dots, X[i]$ 的平均值, $i = 0, \dots, n-1$, 即

$$A[i] = \frac{\sum_{j=0}^i X[j]}{i+1}$$

前缀平均值的计算在经济学和统计学中应用广泛。例如, 给定一个互助基金的年回报, 投资

者想要知道前一年、前三年、前五年以及前十年奖金的年度平均回报。前缀平均值还可以用作某一快速变化的参数的“平滑”函数，如图1-5所示。

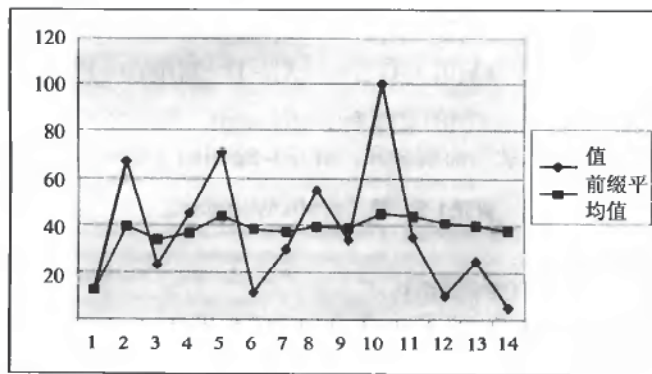


图1-5 阐释前缀平均值函数，以及如何使用它平滑一系列快速变化的值

31

1.4.1 二次时间前缀平均值算法

算法1-4中给出了计算前缀平均值的第一个算法prefixAverages1。算法按照定义分别计算 A 中的每个元素。

算法1-4 算法prefixAverages1

算法 prefixAverages1(X):

输入: 具有 n 个元素的数字数组 X

输出: 具有 n 个元素的数字数组 A , 其中 $A[i]$ 为元素 $X[0], \dots, X[i]$ 的平均值

设 A 是 n 个数字的数组

for $i \leftarrow 0$ to $n-1$ do

$a \leftarrow 0$

 for $j \leftarrow 0$ to i do

$a \leftarrow a + X[j]$

$A[i] \leftarrow a/(i+1)$

return array A

下面分析算法prefixAverages1。

- 算法在开始和结束时分别会初始化和返回数组 A , 其中每个元素都需要执行常数个基本操作。共需 $O(n)$ 时间。
- 有一个嵌套的两层for循环, 分别由计数器 i 和 j 控制。计数器 i 控制外层循环体, 执行 n 次循环, $i=0, \dots, n-1$ 。因此, 每条语句 $a=0$ 和 $A[i]=a/(i+1)$ 都会执行 n 次。这意味着这两条语句以及递增和测试计数器 i 所需的基本操作数与 n 成正比, 即 $O(n)$ 时间。
- 计数器 j 控制内层循环体, 执行 $i+1$ 次循环, 与外层循环计数器 i 的当前值有关。因此, 内层循环中的语句 $a=a+X[j]$ 会执行 $1+2+\dots+n$ 次。回忆定理1.4可知, $1+2+\dots+n=n(n+1)/2$, 这意味着内层循环中的语句的执行时间为 $O(n^2)$ 。通过类似的论证, 可知增加和测试计数器 j 所需的基本操作的执行时间为 $O(n^2)$ 。

将这三项求和得到算法prefixAverages1的运行时间。前两项为 $O(n)$, 第三项为 $O(n^2)$ 。由定理1.2可知, prefixAverages1的运行时间为 $O(n^2)$ 。

32

1.4.2 线性时间前缀平均值算法

为了更高效地计算前缀平均值，可以观察到两个连续平均值 $A[i-1]$ 和 $A[i]$ 是类似的：

$$A[i-1] = (X[0] + X[1] + \cdots + X[i-1]) / i$$

$$A[i] = (X[0] + X[1] + \cdots + X[i-1] + X[i]) / (i+1)$$

如果用 S_i 表示 $X[0] + X[1] + \cdots + X[i]$ 的前缀和(prefix sum)，则前缀平均值为 $A[i] = S_i / (i+1)$ 。用一个循环扫描数组 X ，可以轻松记录当前前缀和。算法1-5给出了详细信息（prefixAverages2）。

算法1-5 算法prefixAverages2

```

算法 prefixAverages2( $X$ ):
  输入: 具有 $n$ 个元素的数字数组 $X$ 
  输出: 具有 $n$ 个元素的数字数组 $A$ ，其中 $A[i]$ 为元素 $X[0], \dots, X[i]$ 的平均值
  设 $A$ 是 $n$ 个数字的数组
   $s \leftarrow 0$ 
  for  $i \leftarrow 0$  to  $n-1$  do
     $s \leftarrow s + X[i]$ 
     $A[i] \leftarrow s / (i+1)$ 
  return array  $A$ 

```

算法prefixAverages2的运行时间分析如下：

- 算法在开始和结束时分别会初始化和返回数组 A ，每个元素都需要执行常数个基本操作。共需 $O(n)$ 时间。
- 开始时，初始化变量 s 所需的时间为 $O(1)$ 。
- 有一个单层for循环，由计数器 i 控制。循环体会执行 n 次， $i = 0, \dots, n-1$ 。因此，语句 $s = s + X[i]$ 和 $A[i] = s / (i+1)$ 都会执行 n 次。这意味着这两条语句以及增加和测试计数器 i 所需的基本操作数与 n 成正比，即需要 $O(n)$ 时间。

对这三项求和可得算法prefixAverages2的运行时间。第一项和第三项为 $O(n)$ ，第二项为 $O(1)$ 。由定理1.2可知，算法prefixAverages2的运行时间为 $O(n)$ 。这个结果比算法prefixAverages1的二次时间要好。

33

1.5 平摊方法

平摊(amortization)是理解算法运行时间的一种重要分析工具。它对于分析每一步性能变化很大的算法很有用。术语“平摊”本身来自会计学领域，它对算法分析提供了一种直观比喻。本节将会看到这一点。

典型数据结构通常支持许多不同的方法用于访问和更新它所存储的元素。同样，某些算法反复迭代，每次迭代执行的工作量不同。在某些情况下，可以根据每个独立操作最坏情况下的运行时间，有效地分析这些数据结构和算法的性能。平摊方法采用的角度不同。它不是单独关注每个操作，而是通过研究一系列操作的运行时间，来考虑所有操作之间的交互。

1. 可清空表数据结构

例如，引入一个简单抽象数据类型(ADT)，即可清空表(clearable table)。这个ADT存储元素的一个表，可通过它们在表中的下标访问这些元素。此外，可清空表还支持以下两个方法：

- **add(e)**: 将元素 e 添加到表中下一个可用单元格。

- `clear()`: 删除表中所有元素以清空表。

设 S 是数组实现的 n 个元素的可清空表, 其大小的固定上界为 N 。操作`clear`需要 $\Theta(n)$ 时间。因为要实际清空表, 需要解除对表中所有元素的引用。

S 是一个初始为空的可清空表, 考虑 S 上的 n 个操作。如果按照最坏情况考虑问题, 单个`clear`操作最坏情况下的运行时间为 $O(n)$, 并且可能有多达 $O(n)$ 个`clear`操作, 可得这 n 个操作的运行时间为 $O(n^2)$ 。虽然这个分析是正确的, 但有些估计过高。这是因为考虑 n 个操作之间的相互影响之后, 整个操作序列的运行时间实际上为 $O(n)$ 。

定理1.10 在一个用数组实现的初始为空的 n 个操作所需时间为 $O(n)$ 。

证明 设 M_0, \dots, M_{n-1} 是 S 上所进行的操作序列, $M_{i_0}, \dots, M_{i_{k-1}}$ 是 k 个`clear`操作, 则有

$$0 \leq i_0 < \dots < i_{k-1} \leq n-1$$

定义 $i_{-1} = -1$ 。由于上一次`clear`操作 $M_{i_{j-1}}$ 或操作序列的开始, 至多有 $i_j - i_{j-1} - 1$ 个元素被添加到表中 (利用`add`操作), 因而操作 M_{i_j} (一次`clear`操作) 的运行时间为 $O(i_j - i_{j-1})$ 。因此, 所有`clear`操作的运行时间为

$$O\left(\sum_{j=0}^{k-1} (i_j - i_{j-1})\right)$$

称这样的求和为叠缩求和 (telescoping sum)。因为除第一项和最后一项外, 其余各项都已相互抵消, 即和为 $O(i_{k-1} - i_{-1})$, 运行时间为 $O(n)$ 。序列中所有其他操作中每个操作所需时间均为 $O(1)$ 。因此, 作用在初始为空的 n 个操作序列的运行时间为 $O(n)$ 。■

定理1.10表明作用在可清空表上任一操作的平均运行时间为 $O(1)$, 平均是指任一操作序列上的平均, 且可清空表初始为空。

2. 平摊一个算法的运行时间

上述例子提供了平摊技术应用的动机, 给出了一种最坏情况下分析平均情况的方法。正式化地讲, 将操作序列最坏情况下的运行时间除以操作个数定义为一个操作的平摊运行时间 (amortized running time)。当操作序列不确定时, 则假设操作序列为某一数据结构的指令表, 且从一个空数据结构开始。因此, 由定理1.10可知, 对于数组实现的可清空表ADT, 每个操作的平摊运行时间为 $O(1)$ 。注意, 一个操作的实际运行时间可能要比它的平摊运行时间长得多 (例如, 某个`clear`操作的运行时间可能为 $O(n)$)。

利用平摊操作的好处在于, 无需利用概率知识, 就可以进行稳健的平均情况分析。它只要求我们具有某种表征用于执行一系列操作的最坏情况运行时间的方式。假定处理整个操作序列所需的实际总时间不超过每个操作给定的平摊界限之和, 甚至可以扩展平摊运行时间的概念, 使得操作序列中的每个操作有自己的平摊运行时间。

有几种方法进行平摊分析。最直接的方法是导出执行一系列操作所需的总运行时间, 这也是在定理1.10的证明中所用的方法。虽然直接方法常用于一系列简单操作中, 利用平摊分析的一些特殊技术, 对这些简单操作进行平摊分析常常较容易。

1.5.1 平摊技术

有两种进行平摊分析的基本技术。一种是基于金融模型的会计方法, 另一种是基于能量模型的势能函数方法。

1. 会计方法

进行平摊分析的会计方法 (accounting method) 利用贷方和借方的模式, 记录序列中不同操作的运行时间。会计方法的基本原理是简单的。可以将计算机看作一台投币设备, 在这个设备中, 一个常量的计算时间需支付一个计算机元 (cyber-dollar)。也可以把一次操作看作一个常量时间的基本操作 (primitive operation) 序列, 每个基本操作的执行需要花费一个计算机元。当执行一个操作时, 应该有足够的计算机元可用于支付它的运行时间。当然, 最直接的方法是对一个操作收取一定数量的计算机元, 这个数量等于所执行的基本操作数。然而, 利用会计方法有趣的一面是, 对操作收取的费用不必是公平的。也就是说, 可以对某些操作过度收费, 而这些操作执行较少的基本操作, 并利用得到的利润帮助那些执行许多基本操作的操作。这种机制可以对序列中的每个操作收取相同数量的计算机元 a , 甚至不会用完支付计算机时间的计算机元。因此, 如果能够建立一种平摊模式 (amortization scheme), 就称序列中每个操作的平摊时间为 $O(a)$ 。当设计一种平摊模式时, 需要考虑将未用完的计算机元存储在数据结构的某些位置中。例如, 存储在表的元素中。

另一种平摊模式是对不同操作收取不同费用。在这种情况下, 一个操作的平摊运行时间是同总费用与操作数的商成正比的。

现在回到可清空表的例子, 并给出它的一种平摊模式, 这产生定理1.10的另一种证明方法。假设一个计算机元足以支付一次访问下标的操作或一次add操作以及clear操作所需要的时间。对每个操作收取两个计算机元。这意味着对clear操作收费较少, 而对其他操作则多收取一个计算机元。以add操作中获得的计算机元将存储在通过操作插入的元素中 (如图1-6所示)。当执行一次clear操作时, 存储在表的每个元素中的计算机元用于支付解除引用的时间。因此, 得到一个有效的平摊模式, 每个操作被收取两个计算机元, 并且所有计算时间都会得到付款。这个简单的平摊模式蕴涵着定理1.10的结论。

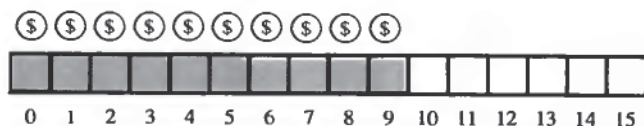


图1-6 表S上操作序列的平摊分析, 计算机元存储在可清空表S的元素中

注意, 当一系列add操作后紧跟一次clear操作时, 会出现最坏情况的运行时间。在其他情况下, 操作序列结束时, 可能还剩余未用完的计算机元。这些计算机元就是那些从下标访问操作获得的计算机元和那些仍然存储在序列的元素中的计算机元。执行 n 个操作序列的计算时间需要支付的计算机元在 $n \sim 2n$ 之间。平摊模式说明, 最坏情况下每个操作需要支付两个计算机元。

在这一点上, 应该强调会计方法是一种简单的分析工具。它不要求修改数据结构或者以任何方式执行算法。尤其不要求增加对象以记录所花费的计算机元。

2. 势能函数

进行平摊分析的另一种有用的技术基于能量模型。在这种方法中, 将结构关联一个值 Φ , 它表示系统的当前能量状态。进行的每次操作都会向 Φ 贡献一个称为平摊时间的额外的量, 但同时也会从 Φ 中抽取与实际花费的时间成比例的一个值。形式上, 设 $\Phi_0 \geq 0$ 表示 Φ 的初始值, 表示系统还未进行任何操作。 Φ_i 表示进行第 i 次操作之后势能函数 Φ 的值。势能函数的主要思想是利用第 i 次操作在势能上的变化 $\Phi_i - \Phi_{i-1}$ 来表征那次操作所需的平摊时间。

以下更详细地分析第 i 次操作的动作。设 t_i 表示它的实际运行时间。定义第 i 次操作的平摊运行

时间为

$$t'_i = t_i + \Phi_i - \Phi_{i-1}$$

即第*i*次操作的平摊开销就是实际运行时间加上那次操作引起的势能变化（可能为正，也可能为负）。换句话说，

$$t_i = t'_i + \Phi_{i-1} - \Phi_i$$

37

即实际花费的时间为平摊时间加上势能净减少量。

设*T'*表示在结构上进行*n*个操作的总平摊时间，即

$$T' = \sum_{i=1}^n t'_i$$

那么*n*个操作花费的总实际时间*T*被限定为

$$\begin{aligned} T &= \sum_{i=1}^n t_i \\ &= \sum_{i=1}^n (t'_i + \Phi_{i-1} - \Phi_i) \\ &= \sum_{i=1}^n t'_i + \sum_{i=1}^n (\Phi_{i-1} - \Phi_i) \\ &= T' + \sum_{i=1}^n (\Phi_{i-1} - \Phi_i) \\ &= T' + \Phi_0 - \Phi_n \end{aligned}$$

上式第二项构成一个叠缩和。换句话说，实际花费的总时间等于总平摊时间加上势能函数在整个操作序列上的净减少量。因此，只要 $\Phi_n \geq \Phi_0$ ，那么 $T \leq T'$ ，实际花费的时间将不会超过平摊时间。

为了使这个概念更具体，利用势能方法对可清空表重复我们的分析。在这种情况下，选择系统的势能 Φ 为可清空表中的实际元素个数。声明任一操作的平摊时间为2，即 $t'_i = 2$ ， $i = 1, \dots, n$ 。为了证明这个结论，考虑第*i*个操作的两种可能的方法。

- **add(*e*)**: 将元素*e*插入表中，使 Φ 增1。实际所需时间为1个时间单位。于是，在这种情况下，

$$1 = t_i = t'_i + \Phi_{i-1} - \Phi_i = 2 - 1$$

显然为真。

- **clear()**: 从表中删除所有*m*个元素所需时间不超过*m*+2个时间单位，包括删除所需的*m*个时间单位和方法调用和开销的至多两个时间单位。这个操作使得系统的势能 Φ 从*m*下降至0（甚至允许*m*=0）。因此，在这种情况下

$$m + 2 = t_i = t'_i + \Phi_{i-1} - \Phi_i = 2 + m$$

显然成立。

因此，在清空表上执行任一操作的平摊时间为 $O(1)$ 。而且，由于对于 $i \geq 1$ ， $\Phi_i \geq \Phi_0$ ，在初始为空的清空表上执行*n*个操作的实际时间*T*为 $O(n)$ 。

38

1.5.2 扩展数组实现分析

上面给出的可清空表ADT的简单数组实现的主要缺点是，对于可能存储在表中的元素总数，要求预先确定固定容量*N*的大小。如果表中的实际元素个数*n*比*N*小得多，那么这种实现就会浪费空间。更糟糕的是，如果*n*的增加超过*N*，这种实现就会崩溃。

我们提供一种工具可以使数组 A 增长,以便存储表 S 中的元素。当然,对于任意常规程序设计语言,如C、C++和Java,实际上是不能使数组增长的,数组的容量为某个固定的数值 N 。当发生溢出(overflow),即 $n = N$ 时,则调用方法add,执行以下步骤:

- (1) 分配一个容量为 $2N$ 的新数组 B 。
- (2) 将 $A[i]$ 复制到 $B[i]$ 中,对于 $i = 1, \dots, N-1$ 。
- (3) 设 $A = B$,即用 B 作为支持 S 的数组。

这个数组替换策略称为可扩展的数组(如图1-7所示)。直觉上,这个策略更像一个寄生蟹,当它生长到超出以前的壳时,会移到另一个较大的壳中。

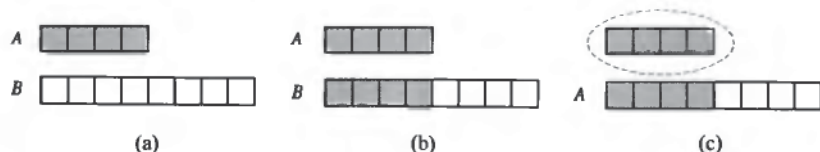


图1-7 增长一个可扩展数组的三步过程:(a)创建新数组 B ; (b)将 A 中的元素复制到 B 中; (c)将对 A 的引用重新分配给新数组。这里没有说明将对原数组进行垃圾收集的过程

从效率的观点看,这种数组替换策略起初似乎是很慢的。由于某些元素插入,需要进行大小为 n 的数组替换,所需时间为 $O(n)$ 。还需注意的是,在进行数组替换之后,所得新数组可以允许向表中插入 n 个新元素,才能再次进行数组替换。这个简单的事实表明,在初始为空的可扩展表上进行一系列操作的运行时间是相当有效的。作为一种简化符号,将作为向量中最后一个元素的元素插入看作一次“add”操作。利用平摊方法,可以证明在一个可扩展数组实现的表上进行一系列add操作,其效率实际上是相当高的。

39

定理1.11 设 S 是可扩展数组 A 实现的表,在 S 上进行 n 个add操作所需的总时间为 $O(n)$,假设开始时 S 为空, A 的大小 $N=1$ 。

证明 利用平摊方法中的会计方法证明这个定理。为了进行分析,再次将计算机看作一台投币设备,且一个常量的计算时间需支付一个计算机元。当执行操作时,在当前的“银行账户”中,应该有足够的计算机元可用于支付那个操作的运行时间。因此,任何计算所花费的计算机元的总量与那次计算所花费的总时间成正比。这种分析的优点在于可以对某些操作过度收费,以便存储起来用于为其他操作付费。

假设 S 中每执行一次add操作需要支付一个计算机元。这不包括扩展数组的时间。假设将数组从 k 增大到 $2k$,需要 k 个计算机元支付复制元素所需的时间。将对每个add操作收取3个计算机元。因此,对于每个add操作多收取两个计算机元并不会引起上溢。考虑一个插入中所剩的两个计算机元,并不像“存储”在插入的元素中那样使数组增长。对于某个整数 $i \geq 0$,表 S 中有 2^i 个元素且可用数组大小为 2^i 时,就会出现上溢。因此,使数组大小加倍需要 2^i 个计算机元。幸运的是,在 $2^{i-1} \sim 2^i - 1$ 个单元中的元素存放了所需的计算机元(如图1-8所示)。注意,当元素个数首次多于 2^{i-1} 时,早先的上溢就会出现,并且存储在 $2^{i-1} \sim 2^i - 1$ 个单元中的计算机元早先还未用尽。因此,平摊模式是有效的,在这种平摊模式中,对每个操作收取3个计算机元,并且这些计算机元支付了所有计算时间。也就是说,可以利用 $3n$ 个计算机元支付 n 个add操作的执行时间。 ■

40

正如描述的那样,随着每次扩展,表的大小可以加倍。我们也可以指定一个显式的capacityIncrement参数,用于确定数组每次扩展应增长的固定量,即将这个参数设置成为一个值 k ,当数组增长时,就将数组增加 k 个单元。必须谨慎利用这个参数。对于大多数应用,正如下

定理所证明的那样，将其大小加倍才是正确的选择。

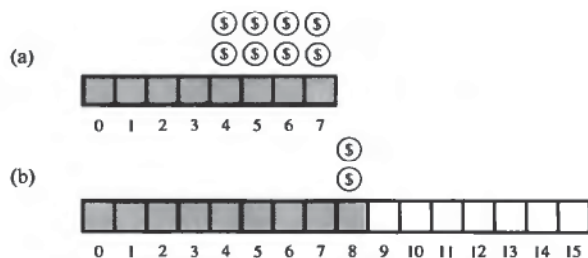


图1-8 昂贵的add操作: (a)8个单元全满, 4~7个单元中每个单元格存储2个计算机元; (b)一个add操作使容量加倍。复制元素需要表中的计算机元, 插入新元素需要对add收取一个计算机元, 获得的两个计算机元存储在单元8中

定理1.12 如果建立一个初始为空的表, 具有固定的正capacityIncrement值, 那么, 在这个向量上进行 n 个add操作所需时间为 $\Omega(n^2)$ 。

证明 设 $c > 0$ 表示capacityIncrement值, $c_0 > 0$ 表示数组的初始大小。当表中当前元素个数为 $c_0 + ic$, $i = 0, \dots, m-1$ 时, 其中 $m = \lfloor (n - c_0)/c \rfloor$, 一个add操作就会引起上溢。因此, 由定理1.4可知, 处理溢出的总时间与下式成正比:

$$\sum_{i=0}^{m-1} (c_0 + ci) = c_0 m + c \sum_{i=0}^{m-1} i = c_0 m + c \frac{m(m-1)}{2}$$

它为 $\Omega(n^2)$ 。因此, 进行 n 个add操作所需时间为 $\Omega(n^2)$ 。 ■

图1-9针对两个初始capacityIncrement值, 比较了add操作序列在初始空表上的运行时间。

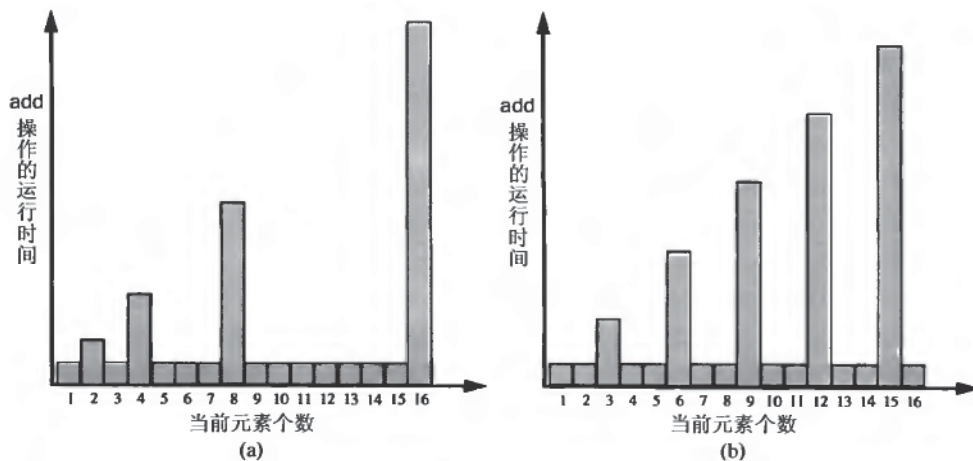


图1-9 可扩展表上add操作序列的运行时间。(a)每次扩展会使数组大小加倍; (b)以capacityIncrement=3进行简单递增

在讨论展开树(3.4节)和集合划分中进行合并寻找(union-find)操作(4.2.2节)的树结构时, 将进一步讨论平摊方法的应用。

1.6 实验

利用渐近分析确定一个算法的运行时间的界限是一个演绎过程。研究算法的伪代码描述。推测关于算法最坏情况的动作，并利用数学工具（如平摊方法、求和以及递归方程）表征算法的运行时间。

这种方法是有效的，但是具有其局限性。渐近分析的演绎方法并不总是能够洞察到算法分析的大O后面隐含的常数因子。同样，对于应该选择一个常数因子较小的慢速算法，还是选择一个常数因子较大的快速算法，演绎方法并没有提供任何指导。此外，演绎方法主要强调算法最坏情况下的输入，这可能并不代表某个问题的典型输入。最后，当一个算法太复杂，而不能有效分析其性能界限时，演绎方法则不适用。

本节讨论一些执行实验算法分析的技术和原理。

1.6.1 实验组织

在进行一项实验的过程中，必须进行若干个步骤才能启动试验。这些步骤需经过深思熟虑，小心执行。

1. 选择问题

启动一项试验要确定的第一件事是决定测试什么。在算法分析领域中，有几种可能性：

- 估计算法的平均渐近运行时间。
- 对于给定的输入值范围 $[n_0, n_1]$ ，测试两个可比算法中哪一个较快。
- 对于带有数值参数（如常量 α 或 ϵ ）的算法，确定它的动作，找出产生最佳性能的参数值。
- 对于试图使某一输入的函数最大化或最小化的算法，测试算法接近最优值的程度。

一旦确定了所需测试的是这些问题甚至其他问题中的哪一个，就能凭经验回答问题，然后开始试验计划的下一步。

2. 确定度量什么

一旦知道要问的问题，接下来集中考虑一些能够回答问题的量化指标。在一个优化问题中，应该度量希望最大化或最小化的函数。对于运行时间，要度量的因素并不是像所想的那样明显。

当然，可以度量算法的实际运行时间。利用过程调用返回一天中的时间，就能度量算法运行之前和之后的时间，将这两个时间相减，可以得到算法运行经过的时间。如果运行的计算机是算法运行所使用的“典型”计算机，这种时间度量是最有用的。

此外，应该认识到运行算法实现时，所谓的“墙上时钟”时间还会受到其他因素的影响，包括计算机上并发运行的程序，算法是否有效地利用了高速缓存，算法是否利用如此之多的内存，使得数据频繁地从二级存储器中换入和换出。所有这些额外的因素都能减慢一个快速算法。因此，如果要用墙上时钟时间度量算法速度，就要确信最小化了这些因素的影响。

度量算法速度的另一种方式采用平台无关性。统计算法中反复使用的基本操作的次数。算法分析中已经有效利用的基本操作的例子包括：

- 内存引用次数。在数据密集的算法中，通过统计内存引用次数得到度量，它将会与算法在任何机器上的运行时间高度相关。
- 比较次数。在算法中，例如排序算法，数据的处理主要是元素之间的比较，因此算法中进行的比较次数与算法的运行时间高度相关。
- 算术操作次数。在数值算法中，算法主要由许多数值计算控制，统计其中的加法次数或乘法次数是算法运行时间的有效度量。这样一个度量可以变成给定计算机上的运行时间，

并对计算机是否有浮点运算协处理器在所获得的性能上进行扩充。

一旦决定想要度量的目标，就必须生成测试数据，并根据测试数据运行算法和收集统计信息。

43

3. 生成测试数据

生成测试数据的目标包含以下几方面：

- 希望生成足够多的样本，以便取平均产生统计意义上的重要结果。
- 希望生成大小可变的输入样本，并能针对各种输入大小，对算法性能进行有根据的推测。
- 希望生成的测试数据具有代表性，也是实际中给定算法的期望数据。

生成足够多且广泛的数据即可满足前两点要求；要生成满足第三点要求的数据，则要经过深思熟虑。需要考虑输入分布，并按照那个分布生成测试数据。随机地简单生成均匀数据并不合适。例如，如果算法基于自然语言文档中可见的单词执行搜索，那么所要求的分布未必是均匀的。理想情况下，希望找到一种方法，收集足够多的实际数据，能够产生统计上有效的结论。当只有部分数据可用时，可以通过随机方法产生一些数据，来满足实际数据的主要统计特性。在任何情况下，应该努力建立测试数据，使我们能够导出支持或是反驳有关算法假设的一般性结论。

4. 编程求解和进行实验

对算法进行正确有效的编程与程序设计技巧有关。而且，如果要将我们的算法与另一个算法进行比较，必须确信这两个算法利用同样水平的程序设计技巧。在这两个待比较的算法实现之间，其代码优化程度应该尽可能接近。对于比较算法要达到一种可比程度无可否认的事实具有主观性，但仍然应该尽可能地努力工作，以便达到这种情况的一种公平比较。最终，我们应该为获得可重现的结果而努力，即具有类似技巧的不同程序员通过进行相似的实验，能够重现同样的结果。

一旦完成程序设计，并生成测试数据，就可实际地进行实验并收集数据。应该在一个尽可能“干净”的环境中进行实验，在数据集合中尽最大可能消除噪声源。应该关注计算环境的细节，包括CPU数量、CPU的速度、主存大小和内存总线速度。

44

1.6.2 数据分析和可视化

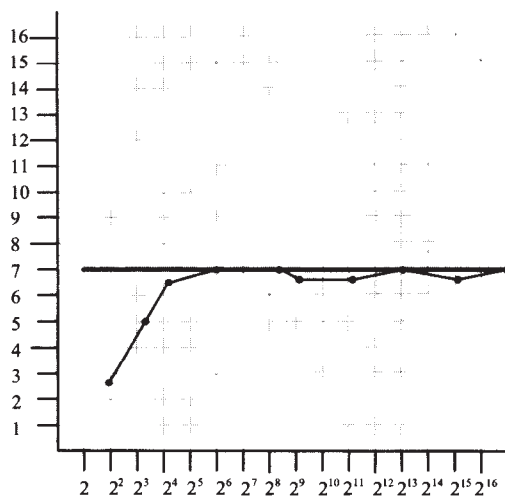
通常在表中观察数据，但这种表示方法常常不像图形那样有用。对于这种数据分析和可视化技术的讨论超出了本书的范围，但在本节中仍然会讨论算法分析的两种有用的分析和可视化技术。

1. 比率测试

在比率测试（ratio test）中，对于某个常量 $c > 0$ ，利用算法知识导出一个关于算法运行时间的主项的函数 $f(n) = n^c$ 。然后设计一种分析法来测试算法的平均时间是否为 $\Theta(n^c)$ 。设 $t(n)$ 表示算法的实际运行时间， n 为某一问题实例的规模。比率测试是利用若干实验中收集的 $t(n)$ 值，绘制出比率 $r(n) = t(n)/f(n)$ 的图形（如图1-10所示）。

如果 $r(n)$ 随着 n 的增长而增长，那么 $f(n)$ 对运行时间 $t(n)$ 估计过低。另一方面，如果 $r(n)$ 收敛到0，那么 $f(n)$ 就会估计过高。但如果比率函数 $r(n)$ 收敛到某一大于0的常量 b ，那么就找到了 $t(n)$ 增长率的一个好的估计值。此外，常量 b 给出了运行时间 $t(n)$ 中的常数因子的一个好的估计值。

尽管如此，应该认识到经验性研究只能测试有限数量的输入和输入大小；因此，比率测试方法不能用于找出指数 $c > 0$ 的精确值。同时它的精确性只限于多项式函数 $f(n)$ 。即便如此，研究表明比率测试方法所能最佳地确定的指数 c 的范围为 $[c - 0.5, c + 0.5]$ 。



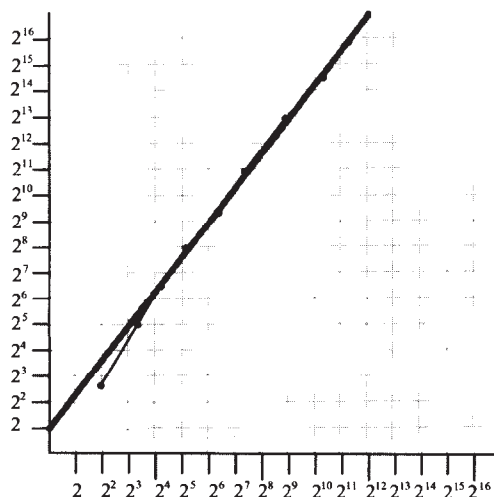
45

图1-10 比率测试估计的示例图，其中 $r(n) = 7$

2. 幂测试

在幂测试中，可以为算法的运行时间 $t(n)$ 产生一个好的估计值，而不是预先为那个运行时间产生一个好的猜测。其想法是首先将实验中收集的、满足 $y = t(x)$ 的数据对 (x, y) ，利用关系式 $(x, y) \rightarrow (x', y')$ 进行转换， x 为样本输入的大小， $x' = \log x$ 和 $y' = \log y$ 。然后画出所有 (x', y') 数据对，并检查结果。

注意，如果对于某些常量 $b > 0$ 和 $c > 0$ ，有 $t(n) = bn^c$ ，那么log-log转换蕴涵着 $y' = cx' + b$ 。因此如果 (x', y') 数据对接近于形成一条直线，那么通过一个简单的直线拟合，就能确定常量 b 和 c 的值。在log-log尺度中，指数 c 对应直线的斜率，系数 b 对应直线的 y 轴截距（如图1-11所示）。另一方面，如果 (x', y') 数据对急剧增长，就能确切地推断出 $t(n)$ 是超多项式，如果 (x', y') 数据对收敛到一个常数，那么 $t(n)$ 可能是亚线性的。在任何情况下，由于可测试的输入大小的有限性，像在比率测试中一样，利用幂测试，很难估计出比范围 $[c - 0.5, c + 0.5]$ 更好的 c 值。

图1-11 幂测试估计值的示例图。在这个例子中，估计 $y' = (4/3)x' + 2$ ；因此估计 $t(n) = 2n^{4/3}$

即便如此,比率测试和幂测试一般还是经验性地估计算法运行时间的好方法。与试图通过降阶技术用多项式拟合测试数据的方法相比,这两种方法相当好。这是因为曲线拟合技术对噪声过度敏感;因此,在多项式运行时间内,它们可能不能给出好的指数估计值。

46

1.7 习题

基础题

- R-1.1 在 x 轴和 y 轴上利用对数作为尺度,画出函数 $12n$ 、 $6n \log n$ 、 n^2 、 n^3 和 2^n 的图形;也就是说,如果函数值 $f(n)$ 为 y ,则将其绘制为一个点,它的 x 坐标为 $\log n$, y 坐标为 $\log y$ 。
- R-1.2 算法 A 利用 $10n \log n$ 个操作,而算法 B 利用 n^2 个操作。确定 n_0 的值,使得其满足对于 $n \geq n_0$,算法 A 好于算法 B 。
- R-1.3 假设 B 利用 $n\sqrt{n}$ 个操作。重做上一个问题。
- R-1.4 证明 $\log^3 n$ 是 $o(n^{1/3})$ 。
- R-1.5 证明以下两种陈述是等价的:
(a)算法 A 的运行时间为 $O(f(n))$ 。
(b)在最坏情况下,算法 A 的运行时间为 $O(f(n))$ 。
- R-1.6 用大 O 符号对以下函数排序。将那些相互是大 Θ 的函数放在一组中(如用下划线)。

$6n \log n$	2^{100}	$\log \log n$	$\log^2 n$	$2^{\log n}$
2^{2^n}	$\lceil \sqrt{n} \rceil$	$n^{0.01}$	$1/n$	$4n^{3/2}$
$3n^{0.5}$	$5n$	$\lfloor 2n \log^2 n \rfloor$	2^n	$n \log_4 n$
4^n	n^3	$n^2 \log n$	$4^{\log n}$	$\sqrt{\log n}$

提示:当拿不准两个函数 $f(n)$ 和 $g(n)$ 时,考虑 $\log f(n)$ 和 $\log g(n)$,或 $2^{f(n)}$ 和 $2^{g(n)}$ 。

- R-1.7 对于下表中的每个函数 $f(n)$ 和时间 t ,确定 t 时间内所求解的最大问题规模。假定求解问题的算法所需时间为 $f(n)$ 微秒。回忆 $\log n$ 表示以2为底的对数。某些条目已经填入。

	1 秒	1 小时	1 个月	1 世纪
$\log n$	$\approx 10^{300\,000}$			
\sqrt{n}				
n				
$n \log n$				
n^2				
n^3				
2^n				
$n!$		12		

47

- R-1.8 Bill有一个算法find2D,用于在一个 $n \times n$ 的数组 A 中查找元素 x 。算法find2D在 A 的每一行上迭代,并调用算法1-3中的算法arrayFind,直到找到 x 或者搜索完 A 的所有行。算法find2D最坏情况下的运行时间是多少?这是一个线性时间的算法吗?为什么?
- R-1.9 考虑以下递归方程, $T(n)$ 定义如下:

$$T(n) = \begin{cases} 4 & \text{如果 } n = 1 \\ T(n-1) + 4 & \text{其他情况} \end{cases}$$

用归纳法证明 $T(n) = 4n$ 。

- R-1.10 根据 n ,用大 O 符号表征算法1-6中所示方法Loop1的运行时间。
- R-1.11 对算法1-6中所示方法Loop2的运行时间进行类似分析。

算法1-6 循环方法集合

```

算法 Loop1( $n$ ):
     $s \leftarrow 0$ 
    for  $i \leftarrow 1$  to  $n$  do
         $s \leftarrow s + i$ 
算法 Loop2( $n$ ):
     $p \leftarrow 1$ 
    for  $i \leftarrow 1$  to  $2n$  do
         $p \leftarrow p \cdot i$ 
算法 Loop3( $n$ ):
     $p \leftarrow 1$ 
    for  $i \leftarrow 1$  to  $n^2$  do
         $p \leftarrow p \cdot i$ 
算法 Loop4( $n$ ):
     $s \leftarrow 0$ 
    for  $i \leftarrow 1$  to  $2n$  do
        for  $j \leftarrow 1$  to  $i$  do
             $s \leftarrow s + i$ 
算法 Loop5( $n$ ):
     $s \leftarrow 0$ 
    for  $i \leftarrow 1$  to  $n^2$  do
        for  $j \leftarrow 1$  to  $i$  do
             $s \leftarrow s + i$ 

```

R-1.12 对算法1-6中所示方法Loop3的运行时间进行类似分析。

R-1.13 对算法1-6中所示方法Loop4的运行时间进行类似分析。

48 R-1.14 对算法1-6中所示方法Loop5的运行时间进行类似分析。

R-1.15 证明如果 $f(n)$ 是 $O(g(n))$ ，且 $d(n)$ 是 $O(h(n))$ ，那么 $f(n) + d(n)$ 之和是 $O(g(n) + h(n))$ 。

R-1.16 证明 $O(\max\{f(n), g(n)\}) = O(f(n) + g(n))$ 。

R-1.17 证明当且仅当 $g(n)$ 是 $\Omega(f(n))$ 时， $f(n)$ 是 $O(g(n))$ 。

R-1.18 证明如果 $p(n)$ 是 n 的多项式，那么 $\log p(n)$ 是 $O(\log n)$ 。

R-1.19 证明 $(n+1)^5$ 是 $O(n^5)$ 。

R-1.20 证明 2^{n+1} 是 $O(2^n)$ 。

R-1.21 证明 n 是 $o(n \log n)$ 。

R-1.22 证明 n^2 是 $\omega(n)$ 。

R-1.23 证明 $n^3 \log n$ 是 $\Omega(n^3)$ 。

R-1.24 证明如果 $f(n)$ 是一个总是大于1的正非降函数，那么 $\lceil f(n) \rceil$ 是 $O(f(n))$ 。

R-1.25 证明如果 $d(n)$ 是 $O(f(n))$ ， $e(n)$ 是 $O(g(n))$ ，那么乘积 $d(n) \cdot e(n)$ 是 $O(f(n)g(n))$ 。

R-1.26 在一个数组实现的初始为空的可扩展表上，如果capacityIncrement参数总是保持为 $\lceil \log(m+1) \rceil$ ， m 为栈中的元素个数，进行 n 次add操作，一个操作的平摊运行时间是多少？即每次表扩展的大小为 $\lceil \log(m+1) \rceil$ 个单元格，扩展后的表容量capacityIncrement为 $\lceil \log(m'+1) \rceil$ ，其中 m 为原表大小， m' 为新表大小（依据当前实际元素个数）。

R-1.27 在一个有 n 个元素的数组 A 中，描述一个查找最小和最大元素的递归算法。你的方法应该返回一个数据对 (a, b) ，其中 a 为最小元素， b 为最大元素。你的方法的运行时间是多少？

R-1.28 假设数组大小从 k 增加到 $2k$ 的开销为 $3k$ 计算机元，重写定理1.11。要使平摊方法有效，应该对每个add操作收取多少计算机元？

R-1.29 以半对数（semi-log）作为尺度，利用比例测试方法，画出点集

$$S = \{(1, 1), (2, 7), (4, 30), (8, 125), (16, 510), (32, 2045), (64, 8190)\}$$

并与以下给定的每个函数进行比较。

- $f(n) = n$ 。
- $f(n) = n^2$ 。
- $f(n) = n^3$ 。

R-1.30 以对数-对数 (log-log) 作为尺度, 画出以下点集。

$$S = \{(1, 1), (2, 7), (4, 30), (8, 125), (16, 510), (32, 2045), (64, 8190)\}$$

利用幂测试方法, 估计多项式函数 $f(n) = bn^c$, 使其最佳地拟合上述数据。

49

创新题

- C-1.1 对于一个有 n 个操作的序列 $P = p_1 p_2 \cdots p_n$, 如果 p_i 的运行时间为 $\Theta(i)$, 且 i 是 3 的倍数, 操作的平摊运行时间为多少? 如果 i 为常数, 则平摊运行时间又为多少?
- C-1.2 设 $P = p_1 p_2 \cdots p_n$ 是 n 个操作序列, 每个操作为 red 操作、或 blue 操作, 且 p_1 是 red 操作, p_2 为 blue 操作。blue 操作的运行时间总是为常数。第一个 red 操作的运行时间为常数, 但在那次运行后的 red 操作 p_i 的运行时间为前一个 red 操作 p_j ($j < i$) 的两倍。在以下条件下, red 操作和 blue 操作的平摊时间为多少?
- 在两个连续的 red 操作之间, 总是有 $\Theta(1)$ 个 blue 操作。
 - 在两个连续的 red 操作之间, 总是有 $\Theta(\sqrt{n})$ 个 blue 操作。
 - 在 red 操作 p_i 和前一个 red 操作 p_j 之间的 blue 操作的个数总是 p_j 和其前一个 red 操作之间 blue 操作个数的两倍。
- C-1.3 对二进制表示的数 $1 \sim n$ 进行计数, 如果将当前的数 i 加 1 的时间与那些必须从 i 变成 $i+1$ 的 i 的二进制表示的位数成正比, 那么总运行时间为多少?
- C-1.4 考虑以下递归方程, 定义函数 $T(n)$:

$$T(n) = \begin{cases} 1 & \text{如果 } n = 1 \\ T(n-1) + n & \text{其他情况} \end{cases}$$

用归纳法证明 $T(n) = n(n+1)/2$ 。

C-1.5 考虑以下递归方程, 定义函数 $T(n)$:

$$T(n) = \begin{cases} 1 & \text{如果 } n = 0 \\ T(n-1) + 2^n & \text{其他情况} \end{cases}$$

用归纳法证明 $T(n) = 2^{n+1} - 1$ 。

C-1.6 考虑以下递归方程, 定义函数 $T(n)$:

$$T(n) = \begin{cases} 1 & \text{如果 } n = 0 \\ 2T(n-1) & \text{其他情况} \end{cases}$$

用归纳法证明 $T(n) = 2^n$ 。

- C-1.7 Al 和 Bill 正在辩论排序算法的性能。Al 声称他的 $O(n \log n)$ 时间的算法总是比 Bill 的 $O(n^2)$ 时间的算法要快。为了解决这个问题, 他们在随机产生的数据集上实现并运行了这两个算法。使 Al 失望的是, 他发现如果 $n < 100$, 实际上 $O(n^2)$ 时间的算法运行较快, 仅当 $n \geq 100$ 时, $O(n \log n)$ 时间的算法更快。解释为什么会出现这种可能。你可以给出数值例子。
- C-1.8 计算机网络中通信安全极其重要, 保证许多网络协议安全的一种方式是对消息加密。在这样的网络上, 安全传输信息的一种典型加密模式是基于这样一个事实, 即不存在有效的大整数因式分解算法。因此, 如果用一个素数 p 表示秘密信息, 可以在网络上传输数 $r = p \cdot q$, $q (> p)$ 是用作加密密钥的另一个素数。在网络上获得传输数 r 的窃听者想要对 r 进行因式分解, 以便计算出加密消息 p 。

50

在不知道密钥 q 的前提下,要进行因式分解计算出消息十分困难。为了理解其原因,考虑以下朴素的因式分解算法:

对于每个满足 $1 < p < r$ 的整数 p ,检查 p 是否能整除 r 。如果可以,则输出“秘密消息是 p !”,算法终止;如果不能整除,则继续。

- a. 假设窃听者利用上述算法,并且有一台计算机,能在1微秒内执行两个多达100位的整数之间的除法运算。如果 r 有100位,给出最坏情况下对加密消息进行解密的时间。
- b. 上述算法最坏情况下的时间复杂度是多少?由于算法输入只是一个大整数 r ,可假设输入大小 n 是存储 r 所需的字节数,即 $n = (\log_2 r)/8$,且每个除法运算所需的时间为 $O(n)$ 。

C-1.9 给出一个正函数 $f(n)$ 的例子,要求满足 $f(n)$ 既不是 $O(n)$,也不是 $\Omega(n)$ 。

C-1.10 证明 $\sum_{i=1}^n i^2$ 是 $O(n^3)$ 。

C-1.11 证明 $\sum_{i=1}^n i/2^i < 2$ 。

提示:尝试用一个几何级数限定这个求和项。

C-1.12 证明 $\log_b f(n)$ 是 $\Theta(\log f(n))$,其中 $b > 1$ 为常数。

C-1.13 描述一个查找 n 个数中最小数和最大数的方法,要求所用的比较次数少于 $3n/2$ 。

提示:首先构造一组候选的最小数和一组候选的最大数。

C-1.14 给定编号为 $1 \sim n$ 的 n 个相同的盒子。前 i 个盒子中每一个中都包含有一粒珍珠,后 $n-i$ 个盒子中是空的。还有两根魔棒,每根都能通过一次接触,确定一个盒子是否为空。如果用魔棒测试空盒子,则它会消失。证明在 i 未知的情况下,你利用两根魔棒至多进行 $o(n)$ 次接触,就能确定所有包含珍珠的盒子。并将魔棒所需接触的次数表示为 n 的渐近函数。

C-1.15 问题同上一题。假设你有 k 根魔棒, $k > 2$ 且 $k < \log n$ 。将识别出包含珍珠的所有魔盒所需的接触次数表示为 n 和 k 的渐近函数。

C-1.16 某个 n 阶多项式形如如下方程

$$p(x) = \sum_{i=0}^n a_i x^i$$

其中 x 为实数, a_i 为常数。

- a. 对于特定的 x 值,描述一个计算 $p(x)$ 的简单的 $O(n^2)$ 时间方法。
- b. 现在考虑将 $p(x)$ 重写为

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n) \cdots)))$$

该式称为Horner规则。利用大O符号,表征这个方法中所用的乘法和加法的次数。

C-1.17 考虑以下归纳“证明”,同一群中的所有羊颜色相同。

归纳基础:一只羊。显然成立。

归纳步骤:有 n 只羊的羊群。从这群羊中取出一只羊 a 。由归纳法,剩下的 $n-1$ 只羊颜色相同。现在把羊 a 放回羊群,再取出另一只羊 b 。由归纳法,剩下的 $n-1$ 只羊颜色相同(羊 a 在其中)。因此, a 与其他羊颜色相同;因此同一群中的所有羊颜色相同。

这个“证明”错在什么地方?

C-1.18 考虑以下“证明”斐波那契函数 $F(n)$ 是 $O(n)$ 的过程。 $F(n)$ 定义如下:

$$F(1) = 1, F(2) = 2, F(n) = F(n-1) + F(n-2)$$

归纳基础($n \leq 2$): $F(1) = 1$ 为 $O(1)$, $F(2) = 2$ 为 $O(2)$ 。

归纳步骤($n > 2$):假设命题对于 $n' < n$ 成立。考虑 n 。 $F(n) = F(n-1) + F(n-2)$ 。由归纳假设 $F(n-1)$ 是 $O(n-1)$, $F(n-2)$ 是 $O(n-2)$ 。那么 $F(n)$ 是 $O((n-1) + (n-2))$ 。由习题R-1.15可知, $O((n-1) + (n-2))$ 是 $O(n)$,因此 $F(n)$ 是 $O(n)$ 。

这个“证明”错在什么地方?

C-1.19 考虑上一个问题中的斐波那契函数 $F(n)$ 。证明 $F(n)$ 是 $\Omega((3/2)^n)$ 。

C-1.20 利用类似于图1-4b中的可视化证明方法,证明 n 为奇数时,定理1.4成立。

- C-1.21 数组 A 中包含 $n-1$ 个 $[0, n-1]$ 内的不同整数, 即所给域中只有一个数不在数组 A 中。设计一个找到这个数的 $O(n)$ 时间的算法。除了数组 A 自身之外, 你只能利用 $O(1)$ 的额外空间。
- C-1.22 证明和 $\sum_{i=1}^n \lceil \log_2 i \rceil$ 是 $O(n \log n)$ 。
- C-1.23 证明和 $\sum_{i=1}^n \lceil \log_2 i \rceil$ 是 $\Omega(n \log n)$ 。
- C-1.24 证明和 $\sum_{i=1}^n \lceil \log_2(n/i) \rceil$ 是 $O(n)$ 。假设 n 为2的幂。
提示: 利用归纳法将问题规模减小到 $n/2$ 。
- C-1.25 一个邪恶的国王有一间存有 n 瓶昂贵美酒的地窖, 他的卫兵刚抓到一个试图向酒投毒的间谍。幸运的是, 卫兵抓到间谍时, 他只向其中一瓶酒投过毒。不幸的是, 他们不知道是哪一瓶。更糟糕的是, 间谍所用的毒药是致命的; 即使把其中一滴稀释到十亿分之一也能杀死人。虽然如此, 这种毒是慢性发作的; 染毒的人需要一个月才会死。设计一个模式使得测试者在一个月内, 至多进行 $O(\log n)$ 次品尝试验, 邪恶的国王就能准确确定哪瓶酒已被投毒。
- C-1.26 设 S 是 n 条直线的集合, 并且任何两条直线都不平行, 任何三条直线都不会通过同一个点。用归纳法证明 S 中的直线确定 $\Theta(n^2)$ 个交点。
- C-1.27 假设 $n \times n$ 数组 A 中的每一行由1或0组成, 并且在 A 的任意一行中, 所有的1都在0之前。假设 A 已在内存中, 描述一个运行时间为 $O(n)$ (不是 $O(n^2)$) 的方法, 找出 A 中包含最多1的行。
- C-1.28 假设 $n \times n$ 数组 A 中的每一行由1或0组成, 并且对于 A 的任意一行 i , 该行中所有的1均在0之前。进一步假设第 i 行中1的个数至少和第 $i+1$ 行中1的个数一样多, $i = 0, 1, \dots, n-2$ 。假设 A 已在内存中, 描述一个运行时间为 $O(n)$ (不是 $O(n^2)$) 的方法, 计算数组 A 中1的个数。
- C-1.29 利用伪代码, 描述一个计算 $n \times m$ 矩阵 A 和 $m \times p$ 矩阵 B 相乘的方法。矩阵乘积 $C = AB$ 定义为: $C[i][j] = \sum_{k=1}^m A[i][k] \cdot B[k][j]$, 你的方法的运行时间是多少?
- C-1.30 只用加法运算, 给出一个计算两个正整数 m 和 n 乘积的递归算法。
- C-1.31 给出新类ShrinkingTable、shrinkToFit的完整伪代码。ShrinkingTable完成可扩展表中的add操作以及remove()操作, 将表中最后一个元素删除。shrinkToFit表示用容量为当前表中元素个数的数组代替当前数组。
- C-1.32 考虑支持add方法和remove方法的可扩展表, 方法定义与上一个问题相同。假定在基础数组需要增长时, 将其容量加倍; 在表中的元素个数降至 $N/4$ 时, 收缩基础数组至原来的一半, N 为数组的当前容量。证明执行 n 个add和remove方法的序列所需时间为 $O(n)$, 初始时 $N=1$ 。
- C-1.33 考虑可扩展表的一种实现, 它不是在容量满时, 把表中元素复制到大小加倍的数组 (即从 N 到 $2N$) 中, 而是把元素复制到有 $\lceil \sqrt{N} \rceil$ 个额外单元的数组中, 容量从 N 到 $N + \lceil \sqrt{N} \rceil$ 。证明在这种情况下, 执行 n 个add操作 (即在尾部进行插入) 序列的时间为 $\Theta(n^{3/2})$ 。

程序设计

- P-1.1 对于1.4节的算法prefixAverages1和prefixAverages2, 进行程序设计。并对它们的运行时间进行仔细的实验分析。并基于线性-线性尺度和对数-对数尺度, 画出运行时间和输入大小关系的曲线。选择有代表性的大小 n 的值, 在测试中, 对于每个 n 至少运行5次测试。
- P-1.2 对于算法1-6中所示的方法, 进行仔细的实验分析, 并比较它们的相对运行时间。利用比率测试和幂测试估计各种方法的运行时间。
- P-1.3 利用数组实现可扩展表, 当增加元素时, 数组大小可增加。对于执行 n 个add方法的序列, 对其运行时间进行实验分析。假设数组大小可从 N 增加到如下值:
- $2N$
 - $N + \lceil \sqrt{N} \rceil$
 - $N + \lceil \log N \rceil$
 - $N + 100$

1.8 本章注记

本章讨论的主题来自多个方面。平摊方法用于分析大量不同数据结构和算法，直到20世纪80年代中期才作为独立专题出现。关于平摊分析的更多信息，请查阅Tarjan的论文[201]，或者Tarjan的书[200]。习题C-1.14受到一个类似问题启发而得，这就是David Ginat研究了从一个高塔上让一个玻璃球急速掉下的问题。

本书中对于大O符号的使用与大部分作者的用法一致。但采用了稍微保守一些的方法。由于大O符号的合理应用[37, 92, 120]，促进了算法与计算理论领域的若干讨论。例如，Knuth[118,120]利用 $f(n) = O(g(n))$ 定义大O符号，虽然他提到大O符号实际上定义了一个函数集合，但是他将其中的“等号”看作是“单向”的。按照Brassard[37]的建议，我们选择将等号看成更标准的形式，把大O符号看作一个集合。对平均情况分析感兴趣的读者可以参考由Vitter和Flajolet[207]所著书籍中的章节。

在附录A中给出了大量有用的数学事实。对研究算法分析感兴趣的读者可参考Graham、Knuth和Patashnik[90]、Sedgewick和Flajolet[184]。有兴趣了解更多数学史的读者可参考Boyer和Merzbach的著作[35]。本书中关于阿基米德的著名故事取自文献[155]。最后，对利用实验估计算法运行时间感兴趣的读者可参阅McGeoch及其合作者的著作[142, 143, 144]的若干篇论文。

第2章



基本数据结构

基本数据结构（如栈和队列）应用广泛。所用的数据结构是否合理往往决定了算法效率的好坏。因此，有必要回顾和讨论一些基本数据结构。

本章首先讨论栈和队列，以及如何用它们来实现递归和多道程序设计。紧接着讨论向量、表和序列抽象数据类型（ADT），每一种数据类型代表一种线性排列的元素集合，并提供访问、插入和删除任意元素的方法。正像栈和队列一样，序列的一个重要性质就是序列中元素的次序由抽象数据类型规范中的操作确定，而不是由元素值确定。

除了这些线性数据结构，本章还将讨论非线性结构。非线性结构利用了丰富的组织关系，而非简单的前后关系。特别要讨论树（tree）抽象数据类型，它所定义的关系是层次（hierarchical）关系，某些对象在一些对象上面，某些对象在一些对象下面。树数据结构的主要术语大多来自族谱，如“双亲”、“孩子”、“祖先”、“后代”是描述层次关系的主要词汇。

本章还要讨论存储“优先元素”的数据结构，每个元素被赋予一个优先级。这个优先级一般是一个数值，但也可将优先级看作任一对象，只要用一致的方法比较这样的对象对即可。优先队列允许选择和删除具有第一优先级的元素。不失一般性，假设该元素为最小元素。采用这种一般的观点可以定义优先队列（priority queue）这样一个一般的抽象数据类型，存储和检索设定优先级的元素。这种ADT本质上不同于本章讨论的基于位置的数据结构，如栈、队列、序列，甚至树。这些数据结构基于特定位置存储元素，这常常就是元素线性排列的位置，这些位置由所进行的插入操作和删除操作决定。优先队列ADT按照元素的优先级存储元素，而没有位置概念。

本章讨论的最后一个数据结构是字典。字典中存储元素，可以利用关键字对元素进行快速定位。进行这种查找的动机是，除了关键字之外，字典中的每个元素还存储了其他一些有用信息。但使用关键字查找是得到这些有用信息的唯一途径。像优先队列一样，字典是一个关键字-元素对的容器。然而，优先队列总是要求关键字满足全序关系，而字典不必满足这种关系。散列表是最简单的字典形式，它假设只能对每个关键字赋予一个整数，并确定两个关键字是否相等。

56

2.1 栈和队列

2.1.1 栈

栈是一个对象容器，插入对象和删除对象按照后进先出（last in first out, LIFO）的原则。在

任何时刻,都可将对象插入栈中,但只有最近(即最后)插入的对象能被删除。栈的名字源于一堆盘子,要取出下面的盘子,必须一个一个将上面的盘子取出,才能取到下面的盘子。在这种情况下,栈的基本操作有“进栈”和“出栈”。

示例2.1 因特网Web浏览器将最近访问的站点地址存储在栈中。每当用户访问一个新网页时,这个网页的地址就被压入地址的栈中。浏览器允许用户利用“返回”按钮回到上次访问的网页。

1. 栈抽象数据类型

栈 S 是一个抽象数据类型,支持以下两个基本方法:

- **push(o):** 在栈顶插入对象 o 。
- **pop():** 删除栈顶对象,并返回该对象,即最近插入的元素仍然在栈中;栈为空时操作出错。

此外,栈还包括如下支持的方法:

- **size():** 返回栈中的对象个数。
- **isEmpty():** 返回布尔值,表明栈是否为空。
- **top():** 返回栈顶对象,但不删除它;栈为空时操作出错。

2. 基于数组的简单实现

可用具有 N 个元素的数组 S 轻松实现一个栈。元素存储在 $S[0] \sim S[t]$ 中,其中 t 为整数,表示 S 中栈顶元素的下标。注意,这样一个实现的重要细节之一是,必须确定栈的最大大小 N ,不妨设为 $N=1000$ (如图2-1所示)。



57

图2-1 用数组 S 实现一个栈。栈顶元素在单元 $S[t]$ 中

回忆本书中对数组的约定,数组的起始下标为0。初始化 t 为-1,并用 t 的这个值判断栈是否为空。同样,用这个变量确定栈中的元素个数($t+1$)。当希望插入一个新元素且数组已满时,必须表明出错条件。给定这个新的异常,算法2-1中实现了栈ADT的主要方法。

算法2-1 用数组实现一个栈

```

算法 push( $o$ ):
    if size() =  $N$  then
        表示栈已满的出错信息
     $t \leftarrow t + 1$ 
     $S[t] \leftarrow o$ 
算法 pop():
    if isEmpty() then
        表示栈已空的出错信息
     $e \leftarrow S[t]$ 
     $S[t] \leftarrow \text{null}$ 
     $t \leftarrow t - 1$ 
    return  $e$ 

```

对于在栈ADT上以常量时间执行栈所支持方法的伪代码描述相当直观。因此,对于用数组实

现的栈，每个方法执行常数条语句，这些语句包括算术运算、比较和赋值。也就是说，在栈ADT的实现中，每个方法的运行时间为 $O(1)$ 。

栈的数组实现既简单又有效，且应用广泛。然而，这种实现也有不利的一面。必须设定栈的最终大小有固定的上界 N 。一个应用实际上可能需要比此更少的空间，在这种情况下，将会浪费内存。另一种情况是，一个应用可能需要比此更多的空间，在这种情况下，只要它试图将第 $N+1$ 个对象插入栈中，栈的实现就可能由于出错而使应用崩溃。因此，即使它的数组实现简明有效，也未必是一种理想的实现。幸运的是，还有其他一些实现方法，将在本章后面讨论。这些实现没有大小限制，所用空间与存储在栈中的实际元素个数成正比。作为另一种方式，还可以利用1.5.2节中讨论过的可扩展表。然而，在可以很好地估计需要入栈的数据项数时基于数组实现的优势仍然难以超越。由于栈在计算应用中起着重要作用，因此快速的栈ADT实现（如数组实现）十分有用。

58

3. 利用栈进行过程调用和递归

栈在现代语言（如C、C++和Java）的运行环境中得到重要应用。在用这种语言所编写的运行程序中，每个线程（thread）都有一个私有栈，称为方法栈（method stack）。在程序执行过程中，方法栈用于记录局部变量的值以及关于方法的其他重要信息（如图2-2所示）。

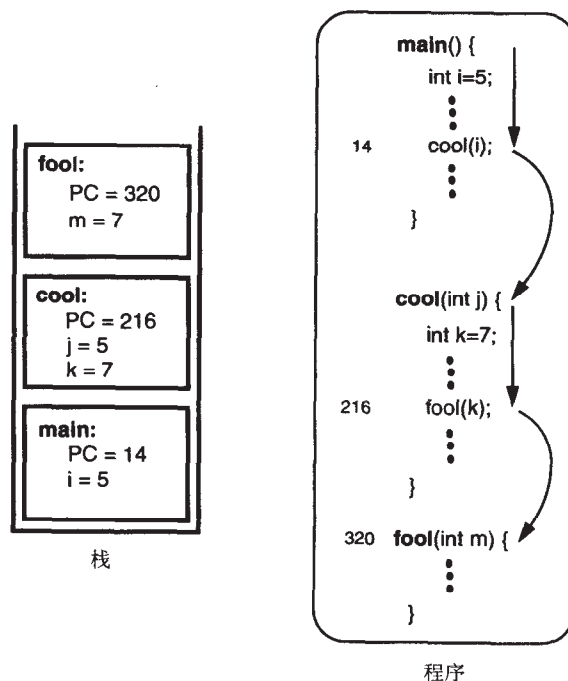


图2-2 方法栈的示例：方法cool调用方法fool，而方法main又调用方法cool。注意存储在栈框架中的程序计数器、参数和局部变量的值。当方法fool的调用终止时，方法cool的调用在指令217处恢复执行，通过递增存储在栈框架中的程序计数器的值获取该值

在程序线程的执行过程中，运行时环境维持一个栈，其元素是当前活动（非终止）的方法调用的描述符。这些描述符称为框架（frame）。调用方法cool的框架存储了方法cool的局部变量的当前值和参数，以及调用cool的方法的信息和需要返回给此方法的信息。

59

运行时环境将程序中线程当前执行的语句的地址存储在一个特殊寄存器中，这个寄存器称为

程序计数器 (program counter)。当方法cool调用另一个方法fool时, 程序计数器中的当前值就被记录在当前方法cool调用的框架中 (因此, 当方法fool执行完成后, 计算机就知道返回到什么地方)。

在方法栈顶是运行方法 (running method) 的框架, 即控制当前执行的方法。栈中其他元素是挂起方法 (suspended method) 的框架。挂起方法是指已经调用另一个方法, 并且等待另一个方法终止, 以将控制权返回给它们的方法。栈中元素次序与当前活动方法调用链有关。当一个新方法被调用时, 此方法的框架就被压入栈中。当它终止时, 它的框架就会从栈中弹出, 并且计算机恢复处理以前挂起的方法。

方法栈也可将参数传递给方法。许多语言 (如C和Java) 的参数利用按值调用 (call-by-value) 传递协议。这表明变量 (或表达式) 的当前值作为参数传递给被调方法。在变量x是基本类型 (如int或float) 的情况下, x的当前值就是与x相关的一个数。当这样的一个值被传递给被调方法时, 这个值就被赋给被调方法框架中的一个局部变量 (图2-2还说明了这个简单的赋值过程)。注意, 如果被调方法改变了这个局部变量的值, 它也不会改变调用方法中的变量值。

4. 递归

利用栈实现方法调用的好处之一是, 允许程序使用递归 (recursion) (1.1.4节), 即使得方法可以调用自身作为子例程。

要正确地利用这个技术, 必须设计一种递归方法, 并能保证递归在某一点终止 (例如, 总是递归调用问题的更小实例, 并作为特例处理最小的实例)。我们注意到, 如果设计一个无限递归方法, 实际上它也不会无限期地运行。在某一点上, 它将用完所有该方法栈的可用内存资源, 产生内存溢出错误。如果小心利用递归, 方法栈将能够毫不费力地实现递归方法。同一方法的每次调用都会关联于一个不同的框架, 包括自己的局部变量的值。递归技术非常强大, 可用于设计简单、有效的程序, 解决相当困难的问题。

60

2.1.2 队列

另一种基本数据结构是队列 (queue), 它是栈的“近亲”。因为队列是一个对象容器, 插入和删除对象按照先进先出 (first in first out, FIFO) 的原则, 即在任何时刻, 都可将对象插入队列中, 但只有插入最长时间 (即最早) 的对象能被删除。通常元素在队尾 (rear) 进入队列, 在队头 (front) 被删除。

1. 队列抽象数据类型

队列ADT将对象保持在一个队列中。对元素的访问和删除限制于队列的第一个元素, 这个元素称为队头 (front)。对元素的插入限制在队列的末端, 称为队尾 (rear)。因此, 规定按照FIFO原则插入和删除队列中的元素。队列ADT支持如下两个基本方法:

- enqueue(o): 将对象o插入队尾。
 - dequeue(): 删除队头对象, 并返回该对象。队列为空时操作出错。
- 此外, 队列ADT包括如下支持的方法:
- size(): 返回队列中的对象个数。
 - isEmpty(): 返回布尔值, 表明队列是否为空。
 - front(): 返回队头元素, 但不删除它。队列为空时操作出错。

2. 基于数组的简单实现

可用具有N个元素的数组Q简单实现一个队列。因为队列ADT按照FIFO原则插入和删除对象, 必须确定如何记录队头和队尾的方式。

一旦将对象放入Q中, 为避免移动它们, 定义两个变量f和r, 意义如下:

- f 是存储队列中第一个元素的 Q 的单元的下标 (也是 `dequeue` 操作要删除的下一个候选项)。如果队列为空, $f=r$ 。
- r 是 Q 中下一个可用数组单元的下标。

61

起初, $f=r=0$, 表明按照条件 $f=r$ 队列为空。当删除队列中的队头元素时, 可以简单地将 f 递增至下一个单元的下标。同样, 当向队列中添加一个元素时, 只要简单地将 r 递增至 Q 中下一个可用单元的下标。不过还需注意不要使得数组上溢。例如, 如果反复对某个元素执行入队和出队操作, 共进行 N 次, 就会有 $f=r=N$ 。此时要再向队列中插入一个元素, 就会发生数组越界错误 (因为 Q 中 N 的有效范围是 $Q[0] \sim Q[N-1]$)。在这种情况下, 即使队列中有足够的空间, 也不能插入元素。为了避免这个问题, 使得能够利用数组 Q 的所有空间, 令 f 和 r 的下标在 Q 的末尾就“绕回”, 即将 Q 看作“循环数组”, 可从 $Q[0]$ 到 $Q[N-1]$, 然后又回到 $Q[0]$, 如图2-3所示。

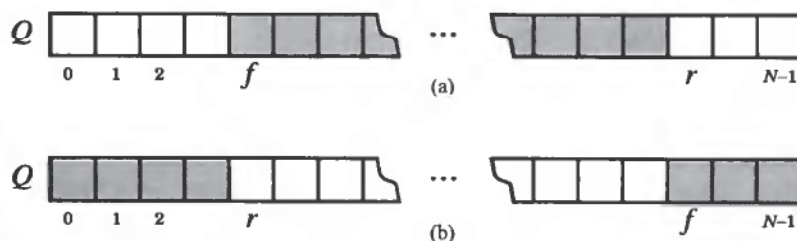


图2-3 循环数组模式: (a) “标准”配置, $f \leq r$; (b) “绕回”配置, $r < f$ 。存储队列元素的单元被加深突出

循环数组 Q 的实现相当简单。每当递增 f 和 r 时, 只需简单地分别计算 “ $(f+1) \bmod N$ ” 或 “ $(r+1) \bmod N$ ”。 \bmod 是取模 (modulo) 算符, 计算结果为整数除法的余数。也就是说, 如果 y 非零, 那么

$$x \bmod y = x - \lfloor x/y \rfloor y$$

考虑这样一种情况, 向队列中插入 N 个对象, 而不进行删除操作。这时 $f=r$ 。这和队列为空时的情况一样。因此, 在这种情况下, 不能区分队列是满还是空。幸运的是, 这不是一个大问题, 有多种方法可以处理这种情况。例如, 可以简单设置, 使 Q 不能存放多于 $N-1$ 个对象。上述处理队列满时的简单规则兼顾了实现的最终问题。算法2-2给出了队列的主要方法的伪代码描述。注意, 利用表达式 $(N-f+r) \bmod N$ 计算队列大小, 这种计算方式可以将“正常”配置 $f \leq r$ 和“绕回”配置 $r < f$ 统一起来。

62

算法2-2 队列的循环数组实现

```

算法 dequeue():
    if isEmpty() then
        throw a QueueEmptyException
    temp ← Q[f]
    Q[f] ← null
    f ← (f+1) mod N
    return temp
算法 enqueue(o):
    if size() = N-1 then
        throw a QueueFullException
    Q[r] ← o
    r ← (r+1) mod N

```

如同基于数组的栈实现，数组实现的每个队列方法执行常数条语句，这些语句包括算术运算、比较和赋值。因此，该实现中的每个方法的运行时间为 $O(1)$ 。

同样，如同基于数组的栈实现，基于数组的队列实现的唯一缺点是人为地设置了队列容量为某个数 N 。在实际应用中，所需空间实际上可能比设置的要大或要小一些。但如果能够同时对要放入队列中的元素个数进行良好的估计，那么基于数组的实现是相当有效的。

3. 利用队列进行多道程序设计

多道程序设计（multiprogramming）是获得有限并行性的一种方法，即使在计算机上只有一个CPU也是如此。这种机制允许同时运行多个任务或计算线程（thread），每个线程负责某一特定计算。多道程序设计广泛应用于图形应用中。例如，一个线程可以负责捕获鼠标单击，而其他线程则负责屏幕周围动画的移动部分。即使计算机只有一个CPU，这些不同的计算线程也会看起来在同时运行。这是因为：

- 相对于我们对时间的感觉，CPU运行速度很快。
- 操作系统为每个线程提供不同的CPU“时间片”。

63

为每个不同的线程分配的时间片快速连续出现，以至于好像不同的线程在同时/并行运行一样。

例如，Java有一种内嵌机制——Java线程，可以实现多道程序设计。Java线程是一些计算对象，它们可以相互合作和通信，共享内存中的对象、计算机的屏幕，或其他类型的资源 and 设备。Java程序中不同线程之间的切换相当快速，因为每个线程在Java虚拟机的内存中都存储有自己的Java栈。在每个线程的方法栈中，包含当前正在运行的那个线程的方法的局部变量和框架。因此，要从线程 T 切换到另一个线程 U ，CPU所需做的是在切换到线程 U 之前，记住离开线程 T 的位置。我们已经讨论通过存储 T 的程序计数器中的当前值，完成切换的一种方式，这个当前值是线程 T 要执行的下一条指令的引用，它存储在 T 的Java栈顶。通过将每个活动线程的程序计数器的值存储在其Java栈顶，并通过存储在 U 的Java栈顶的值，恢复程序计数器的值（将 U 的栈用作“当前”Java栈），CPU在另一个线程 U 中就能知道它离开的位置。

当设计利用多线程的程序时，必须仔细考虑，不允许某个线程独占CPU时间。这样的独占可能导致应用程序或小程序挂起（hanging），它在技术上是运行的，但实际上不做任何事情。在某些操作系统中，线程独占CPU不是一个问题，然而，这些操作系统根据循环（round-robin）协议，利用一个队列将CPU时间分配给可运行线程。

循环协议的主要思想是将所有可运行线程存储在队列 Q 中。当CPU准备向一个线程提供时间片时，对 Q 执行dequeue操作，获得下一个可用的可运行线程；称这个线程为 T 。但是，在CPU开始执行 T 的指令之前，它会启动一个运行在硬件集中的定时器，该定时器被设置成在一段固定的时间之后到期。CPU现在等待直到(a)线程 T 阻塞自己，或(b)定时器到期。在后一种情况下，CPU停止线程 T 的执行，并执行enqueue操作，将线程 T 置于当前可运行线程队列的队尾。不管是哪一种情况，CPU都将 T 的程序计数器的当前值存储在 T 的方法栈的栈顶，并处理从队列 Q 中提取的下一个可用的可运行线程。这样，CPU保证每个可运行线程具有公平的时间共享。因此，利用简单队列数据结构和硬件停表，操作系统就能避免CPU被独占。

虽然这个基于队列的方法解决了多道程序设计问题，还应注意到，这种解决方法实际上是对大多数操作系统所用协议的一种过度简化，在这些操作系统中，采用循环方式分配时间片，因为大多数系统给出了线程优先级。因此，它们利用优先队列（priority queue）实现时间片。将在2.4节讨论优先队列。

64

2.2 向量、表和序列

栈和队列按照线性序列存储元素，该序列由在其“末端”作用的更新操作确定。本节讨论的数据结构维持线性顺序，同时允许在序列的“中间”进行访问和更新。

2.2.1 向量

假设 S 是一个包含 n 个元素的线性序列。可以利用一个 $[0, n-1]$ 内的整数唯一指定 S 中的每个元素 e ，这个整数等于 S 中元素 e 之前的元素个数。定义 S 中元素 e 的位序(rank)为其之前的元素个数。因此，序列中第一个元素的位序为0，最后一个元素的位序为 $n-1$ 。

注意，位序类似于数组下标，但并不一定利用数组实现一个队列，在这种情况下，位序为0的元素存储在数组中的下标0处。位序定义提供了一个引用序列中元素下标的方式，而不需考虑表的精确实现问题。注意当序列被更新时，元素的位序可能发生变化。例如，如果在序列的起始处插入一个新元素，则其他每一个元素的位序都会增1。

支持通过位序访问其元素的线性序列称为向量(vector)。位序的表示简单有效，因为它可用于确定在向量的什么位置插入一个新元素，以及在什么位置删除一个旧元素。例如，可以给出新元素插入之后的位序(例如，在位序2处插入)。也可利用位序确定要删除的元素(例如，删除位序2处的元素)。

1. 向量抽象数据类型

存储 n 个元素的向量(vector) S 支持以下基本方法：

- $\text{elemAtRank}(r)$ ：返回 S 中位序 r 处的元素；如果 $r < 0$ 或 $r > n-1$ 则出错。
- $\text{replaceAtRank}(r, e)$ ：用 e 替换位序 r 处的元素，并返回该元素；如果 $r < 0$ 或 $r > n-1$ 则出错。
- $\text{insertAtRank}(r, e)$ ：在 S 中插入位序为 r 的新元素 e ；如果 $r < 0$ 或 $r > n$ 则出错。
- $\text{removeAtRank}(r)$ ：删除 S 中位序为 r 的元素；如果 $r < 0$ 或 $r > n-1$ 则出错。

65

此外，向量支持常用方法 $\text{size}()$ 和 $\text{isEmpty}()$ 。

2. 基于数组的简单实现

可用数组 A 实现向量ADT，其中 $A[i]$ 存储(引用)位序为 i 的元素。数组 A 的大小 N 选择得足够大，将向量中的元素个数 $n < N$ 保持在实例变量中。向量ADT的方法的实现细节比较简单。例如，为了实现 $\text{elemAtRank}(r)$ 操作，只需返回 $A[r]$ 。算法2-3中给出了方法 $\text{insertAtRank}(r, e)$ 和方法 $\text{removeAtRank}(r)$ 的实现。该实现的主要(且耗时)的部分涉及上移或下移元素以保持所占据数组单元的连续性。这些移动操作需要维持位序为 i 的元素存储在数组 A 的下标 i 处。如图2-4所示，另请参见习题C-2.5。

算法2-3 向量ADT的数组实现中的方法

```

算法  $\text{insertAtRank}(r, e)$ :
    for  $i = n-1, n-2, \dots, r$  do
         $A[i+1] \leftarrow A[i]$  {为下一个元素腾出空间}
     $A[r] \leftarrow e$ 
     $n \leftarrow n+1$ 

算法  $\text{removeAtRank}(r)$ :
     $e \leftarrow A[r]$  { $e$ 是一个临时变量}
    for  $i = r, r+1, \dots, n-2$  do
         $A[i] \leftarrow A[i+1]$  {填充被删除的元素}
     $n \leftarrow n-1$ 
    return  $e$ 
  
```

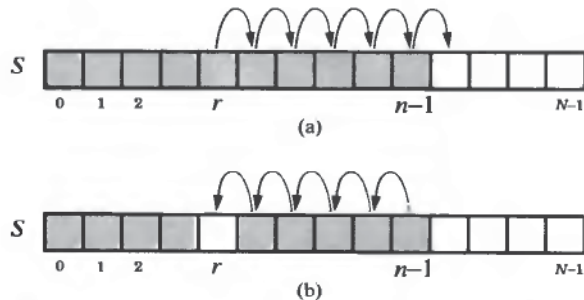


图2-4 向量 S 基于数组的实现，其中存储 n 个元素：(a)在位序 r 处插入时向上移动；
(b)在位序 r 处删除时向下移动

66

表2-1表明了数组实现的向量的方法的运行时间。方法`isEmpty`、`size`和`elemAtRank`的运行时间显然为 $O(1)$ 。但插入和删除方法的运行时间要比这长得多。尤其是，`insertAtRank(r , e)`在最坏情况下的运行时间为 $\Theta(n)$ 。当 $r = 0$ 时，这个操作就会出现最坏情况，此时现有的 n 个元素都必须向前移动。类似的情况出现在方法`removeAtRank(r)`中，它的运行时间为 $O(n)$ ，在最坏情况($r = 0$)下要向后移动 $n-1$ 个元素。实际上，假设每个可能的位序等概率作为参数传递给这些操作，则它们的平均运行时间为 $\Theta(n)$ ，因为平均需要移动 $n/2$ 个元素。

表2-1 数组实现的具有 n 个元素的向量在最坏情况下的性能。占用空间为 $O(N)$ ，其中 N 为数组大小

方 法	时 间
<code>size()</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$
<code>elemAtRank(r)</code>	$O(1)$
<code>replaceAtRank(r, e)</code>	$O(1)$
<code>insertAtRank(r, e)</code>	$O(n)$
<code>removeAtRank(r)</code>	$O(n)$

更仔细地观察`insertAtRank(r , e)`和`removeAtRank(r)`，可见它们中的每一个方法的运行时间为 $O(n-r+1)$ ，因为只有在位序 r 和更大位序处的元素需要被上下移动。因此，利用方法`insertAtRank(n , e)`和`removeAtRank($n-1$)`，在向量末尾插入或删除一个元素所需时间分别为 $O(1)$ 。也就是说，在向量末尾插入或删除一个元素需要常量时间。在距向量末尾常数个单元内插入和删除元素也会需要常量时间。然而，基于上述实现，在向量的开始插入或删除元素需将所有其他的元素移动一次；这需要 $\Theta(n)$ 时间。因此，这个向量实现存在不对称性——在末尾更新较快，在开始更新较慢。

实际上，稍作努力，就能产生一个数组实现的向量ADT，在位序0处进行插入和删除操作的时间为 $O(1)$ ，与在末尾进行插入和删除操作的时间一样。要达到这个要求，就要放弃规定：即位序为 i 的元素存储在数组的下标 i 处。可以利用2.1.2节中的方法，利用循环数组实现队列。我们将实现的细节留作习题(C-2.5)。此外，也可用可扩展表(1.5.2节)有效地实现向量，这实际上是Java中向量ADT的默认实现。

67

2.2.2 表

利用位序不仅意味着引用元素在表中出现的位置。可用另一种方式实现表 S ，使得每个元素存储在一个特殊结点(node)对象中，并引用表中它之前和之后的结点。在这种情况下，利用结点比位序要自然和有效一些，它可用于确定访问和更新表的位置。本节研究用结点概念抽象描述

表中“位置”的方式。

1. 位置和基于结点的操作

我们想要定义表 S 的方法，这些方法将表中的结点作为参数，并作为返回类型提供它们。例如，定义一个假想方法 $\text{removeAtNode}(v)$ ，它将 S 中存储在表中结点 v 处的元素删除。利用结点作为参数，可以用 $O(1)$ 时间删除一个元素，这只需直接到达存储结点的地方，然后通过更新其近邻指针的引用，断开这个结点的连接。

通过增加这样一个基于结点的操作，定义表ADT的方法提出了一个问题，这就是在表的实现中应该放置多少信息。可以肯定的是，希望能够利用这样一个实现，而不向用户展示细节。同样，也不允许用户在不知情的情况下修改表的内部结构。为了对表的各种实现进行抽象，并统一存储元素的不同方式，在表中引入位置（position）的概念，这就形式化了表中一个元素相对于其他元素的“位置”的概念。

为了安全地展开表的操作集，将位置概念进行抽象，这使我们可以享受基于结点表实现的效率，而不违背面向对象的设计原则。在这种框架中，将表看作元素的容器，它将每个元素存储在一个位置上，然后将这些位置按线性顺序排列。位置自身是一种抽象数据类型，它支持以下简单方法：

- $\text{element}()$ ：返回存储在该位置的元素。

位置的定义总是相对的（relatively），即根据它的近邻定义。在一个表中，位置 p 总是位于某个位置 q “之后”和某个位置 s “之前”（除非 p 是第一个位置或最后一个位置）。除非显式删除 e （因而破坏了位置 p ），否则即使表 S 中 e 的位序改变，关联于该元素 e 的位置 p 也不会改变。此外，即使用另一个元素替换或交换存储在位置 p 处的元素 e ，位置 p 也不会改变。这些关于位置的事实允许定义大量基于位置的表方法集合，它们将位置对象作为参数，并且还作为返回值提供位置对象。

68

2. 表抽象数据类型

利用位置概念将结点的思想封装在表中，可以定义另一种序列ADT，简单称其为表（list）ADT。这种ADT支持表 S 的如下方法：

- $\text{first}()$ ：返回 S 中第一个元素的位置；如果 S 为空则出错。
- $\text{last}()$ ：返回 S 中最后一个元素的位置；如果 S 为空则出错。
- $\text{isFirst}(p)$ ：返回一个布尔值，表明给定位置是否是表的第一个位置。
- $\text{isLast}(p)$ ：返回一个布尔值，表明给定位置是否是表的最后一个位置。
- $\text{before}(p)$ ：返回 S 中位置 p 之前的位置；如果 p 是第一个位置，则出错。
- $\text{after}(p)$ ：返回 S 中位置 p 之后的位置；如果 p 是最后一个位置，则出错。

上述方法允许引用表中的相对位置，不论从起始位置或末尾位置开始，都能够在表中来回移动。可以直观地将这些位置看作表中的结点，但需注意在这些方法中，没有对结点对象的特定引用，也没有指向上一结点或下一结点的链接。除了上述方法和一般方法 size 和 isEmpty 之外，表ADT还包括以下更新方法。

- $\text{replaceElement}(p, e)$ ：用 e 替换位置 p 处的元素，返回替换之前位置 p 处的元素。
- $\text{swapElements}(p, q)$ ：交换存储在位置 p 、 q 处的元素，使得位置 p 处的元素移到位置 q ，位置 q 处的元素移到位置 p 。
- $\text{insertFirst}(e)$ ：向 S 中插入新元素 e ，作为第一个元素。
- $\text{insertLast}(e)$ ：向 S 中插入新元素 e ，作为最后一个元素。
- $\text{insertBefore}(p, e)$ ：在表 S 中的位置 p 之前插入新元素 e 。

69

- `insertAfter(p, e)`: 在表 S 中的位置 p 之后插入新元素 e 。
- `remove(p)`: 删除表 S 中位置 p 处的元素。

表ADT可以根据对象的位置, 将对象看作有序集合, 而不用考虑这些位置的精确表示方式。同时, 需注意的, 在上述表ADT的方法中, 有一些冗余。也就是说, 可以通过检查 p 是否等于`first()`的返回位置, 执行操作`isFirst(p)`。也可以通过执行操作`insertBefore(first(), e)`, 执行操作`insertFirst(e)`。这些冗余的方法应该被看作捷径。

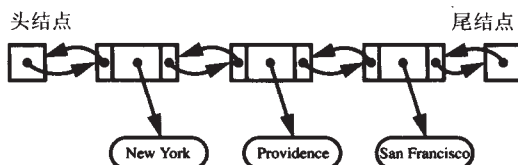
如果将位置作为参数传递给表操作之一无效, 就会出现错误。位置 p 无效的原因包括它为`空`, 它是另一个表的位置或者是以前从表中删除的位置。

具有内置位置概念的表ADT大量用于设置中。例如, 简单文本编辑器嵌入有位置插入和删除的概念。因为这样的编辑器通常会执行所有与光标 (cursor) 有关的更新操作, 而光标表示被编辑文本字符在表中的当前位置。

3. 链表实现

可以对链表 (linked list) 数据结构进行大量操作, 包括在各个地方进行插入和删除, 其运行时间为 $O(1)$ 。单链表 (singly linked list) 中的一个结点在`next`链域中存放指向表中下一个 (`next`) 结点的引用。因此, 只能在从头至尾这一个方向上遍历单链表。而双向链表 (doubly linked list) 中的结点存放两个引用, 在`next`链域中存放指向表中下一个结点的引用, 在`prev`链域中存放指向表中前一个结点的引用。因此, 可以在两个方向上遍历双向链表。对于双向链表, 可以从任何给定的结点确定它的前一结点和下一结点, 极大地简化了表的实现。因而, 假设用双向链表实现表ADT。

为了简化更新和查询操作, 在表的两端分别增加一个特殊结点: 即在表头之前的头结点 (`header`) 和在表尾之后的尾结点 (`trailer`)。在这些虚拟结点或是观察哨 (`sentinel`) 结点中并不存储任何元素。头结点有一个有效的`next`引用域, 但其`prev`引用域为`空`; 而尾结点有一个有效的`prev`引用域, 但其`next`引用域为`空`。图2-5中给出了带有观察哨的双向链表。注意一个链表对象只需存储观察哨以及`size`计数器的内容, 用来记录表中的元素个数 (不统计观察哨)。



70

图2-5 带有观察哨、头结点和尾结点的双向链表, 标出了表尾。当观察哨相互指向时, 表为`空`

可以简单地用链表中的结点实现位置ADT, 定义一个方法`element()`, 用于返回存储在结点中的元素。因此, 结点自身用作位置。

如果要在位置 p 后面插入一个元素 e , 考虑如何实现`insertAfter(p, e)`方法。首先创建一个新结点 v 存放元素 e , 并将 v 链接进表中其位置, 然后更新 v 的两个新近邻的`next`引用和`prev`引用。算法2-4中给出了该方法的伪代码描述, 图2-6给出了图示说明。

算法2-4 在链表的位置 p 后面插入元素 e

```

算法 insertAfter(p, e):
    创建一个新结点 $v$ 
     $v.\text{element} \leftarrow e$ 
     $v.\text{prev} \leftarrow p$            { $v$ 链接到它的直接前驱}
  
```



```

v.next ← p.next    {v链接到它的直接后继}
(p.next).prev ← v   {将p的原后继链接到v}
p.next ← v          {将p链接到它的新后继v}
return v            {返回元素e的位置}

```

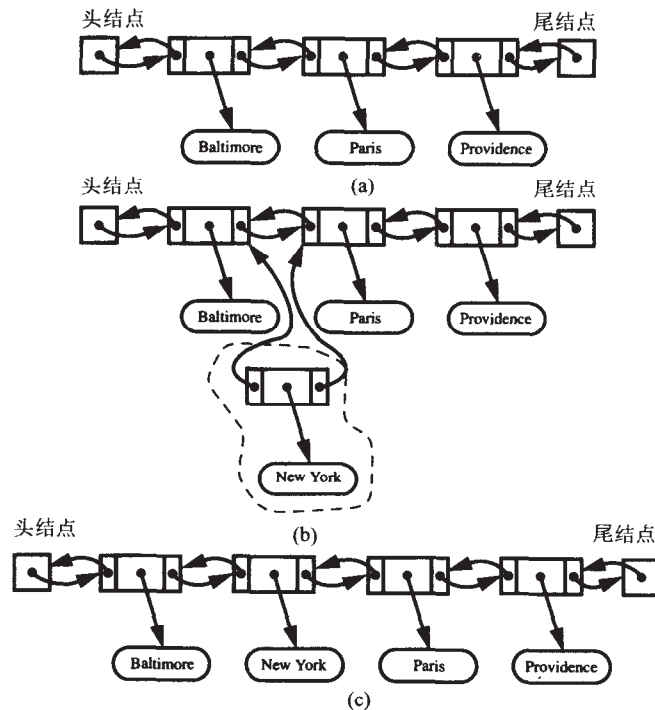


图2-6 在元素“Baltimore”的位置后面添加一个新结点：(a)插入前；(b)创建结点 v 并将其链接到链表中；(c)插入后

71

方法insertBefore、insertFirst和insertLast的算法类似于方法insertAfter的算法。将它们留作习题(R-2.1)。下面考虑remove(p)方法，它用于删除存储在位置 p 处的元素 e 。为进行这个操作，将 p 的两个近邻相互链接成彼此引用的新近邻。这样就将 p 排除在链外（即删除）。注意，在将 p 排除在链外之后，没有结点指向 p ；因此，垃圾收集器可以收回为 p 分配的空间。算法2-5中给出了该方法的算法，图2-7给出了图示说明。回忆对观察哨的使用，注意到不论 p 是表中第一个元素、最后一个元素，或者只是表中的真实位置，这个算法都有效。

算法2-5 在链表中删除存储在位置 p 处的元素 e

```

算法 remove( $p$ ):
     $t \leftarrow p.element$            {临时变量，用于保存返回值}
    ( $p.prev$ ).next ←  $p.next$        {删除 $p$ }
    ( $p.next$ ).prev ←  $p.prev$ 
     $p.prev \leftarrow null$          {使位置 $p$ 无效}
     $p.next \leftarrow null$ 
    return  $t$ 

```

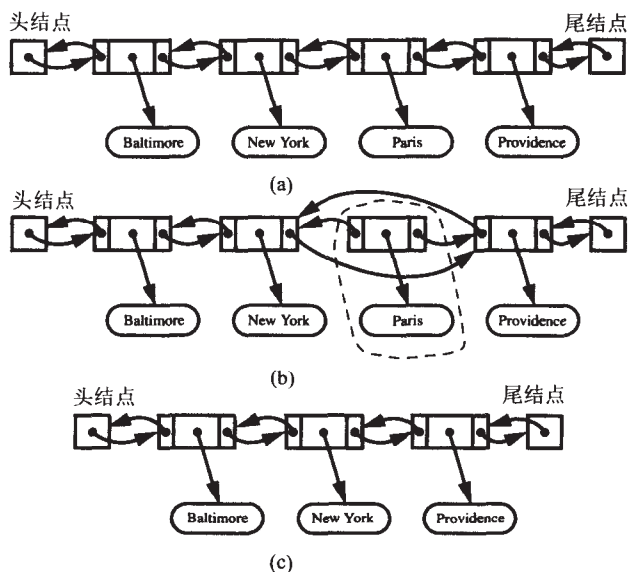


图2-7 删除存储在“Paris”位置的对象：(a)删除前；(b)将原结点排除在链外；(c)删除（并进行垃圾收集）后

72

2.2.3 序列

本节定义广义序列ADT，它包括向量ADT和表ADT的所有方法。这个ADT提供了用位序和位置访问元素的方式，是一种通用数据结构，应用广泛。

1. 序列抽象数据类型

序列（sequence）是一种ADT，支持向量ADT（在2.2.1节讨论）和表ADT（在2.2.2节讨论）的所有方法，还支持连接位序和位置的以下两个“桥接”方法：

- $\text{atRank}(r)$ ：返回位序为 r 的元素的位置。
- $\text{rankOf}(p)$ ：返回位置 p 处的元素的位序。

可用双向链表或数组实现广义序列，需在这两种实现之间进行自然的折中。表2-2比较了通过数组（利用循环方式）和双向链表实现的广义序列ADT的运行时间。

表2-2 用数组（利用循环方式）和双向链表实现的序列的方法的运行时间比较。 n 表示执行操作时序列中的元素个数。双向链表实现占用空间为 $O(n)$ ，数组实现占用空间为 $O(N)$ ，其中 N 为数组大小

操 作	数 组	表
size、isEmpty	$O(1)$	$O(1)$
atRank、rankOf、elemAtRank	$O(1)$	$O(n)$
first、last、before、after	$O(1)$	$O(1)$
replaceElement、swapElements	$O(1)$	$O(1)$
replaceAtRank	$O(1)$	$O(n)$
insertAtRank、removeAtRank	$O(n)$	$O(n)$
insertFirst、insertLast	$O(1)$	$O(1)$
insertAfter、insertBefore	$O(n)$	$O(1)$
remove	$O(n)$	$O(1)$

由表2-2可知,对于基于位序的访问操作(atRank、rankOf和elemAtRank),基于数组的实现优于链表实现。对于所有其他访问操作,这两种实现性能相同。对于基于位置的更新操作(insertAfter、insertBefore和remove),链表实现优于基于数组的实现。即便如此,对于基于位序的更新方法(insertAtRank和removeAtRank),这两种实现最坏情况下的性能相同,但原因不同。在更新操作(insertFirst和insertLast)中,这两种实现也有可比较的性能。

73

考虑空间使用情况。一个数组需要 $O(N)$ 的空间,其中 N 为数组大小(除非利用可扩展数组)。而双向链表利用 $O(n)$ 的空间,其中 n 为序列中的元素个数。因为 n 小于或等于 N ,这意味着链表实现所用的空间渐近优于固定大小的数组所用的空间,尽管由于数组不需要链接来维持其单元的次序,而链表需要较大的常数因子开销。

数组实现和链表实现都有其优缺点。对于某一应用应该选择哪一种实现方法,取决于该应用要进行何种操作和需要的存储空间。设计序列ADT的方式并不依赖于其实现方式。因此只需对程序进行不多的修改,就能容易地在这两种实现之间进行切换,使得实现最好地适合于应用。

2. 迭代器

向量、表或序列上的典型计算是通过它的元素次序进行的,一次一个计算。例如,查找特定的元素。

迭代器(iterator)是一种软件设计模式,它抽象了一次一个元素地扫描元素集合的过程。迭代器包含序列 S 、 S 中的当前位置和步进到 S 中下一位置的方式并使之成为当前位置。因此,迭代器扩展了2.2.2节介绍的位置ADT的概念。事实上,可将位置看作静止的迭代器。迭代器将“位置”和“下一个”的概念封装在一个对象集合中。

我们将迭代器ADT定义为支持以下两个方法:

- hasNext: 测试迭代器中是否还有剩余的元素。
- nextObject: 返回并删除迭代器中的下一个元素。

注意这个ADT在遍历一个序列时利用“当前”元素的概念。假设迭代器中至少包含一个元素,首次调用方法nextObject将返回迭代器中的第一个元素。

迭代器为访问一个容器(对象集合)中的所有元素提供了一致的框架。在某种程度上,它不依赖于特定集合组织。序列的迭代器应该按照元素的线性顺序返回元素。

74

2.3 树

树是一种分层存储元素的抽象数据类型。除了顶层元素之外,树中的每个元素都有一个父结点(parent)元素,0个或多个子结点(child)元素。通常将元素放在椭圆或矩形框中,并用直线将这些父、子元素连接起来,以使树可视化。如图2-8所示。称顶层元素为树的根(root),但通常将其绘制为最高层的元素,其他元素连在其下面(正好和植物界的树相反)。

树(tree) T 是按照父-子关系存储元素的结点(node)集合,具有如下性质:

- T 有一个特殊结点 r ,称为 T 的根。
- T 中每个非根结点 v 有一个父结点 u 。

按照上述定义,树不能为空。因为它至少有一个根结点。可以扩充树的定义使其包括空树。但仍然采用常规约定,即认为树总是有一个根,避免在算法中总是处理空树的特殊情况,使得陈述简单。

如果结点 u 是结点 v 的父结点,那么称 v 是 u 的子(child)结点。具有同一父结点的两个结点称为兄弟(sibling)。如果一个结点没有子结点,则称它是外部结点(external),如果一个结点有一

个或多个子结点，则称它是内部结点 (internal)。外部结点也称为叶结点 (leave)。根为 v 的树的子树也是一棵树，由 v 的所有后代组成 (包括 v 自身)。结点的祖先 (ancestor) 要么为结点自身，要么为该结点的父结点的祖先。相反，如果结点 u 是结点 v 的一个祖先，则称结点 v 是结点 u 的后代 (descendent)。

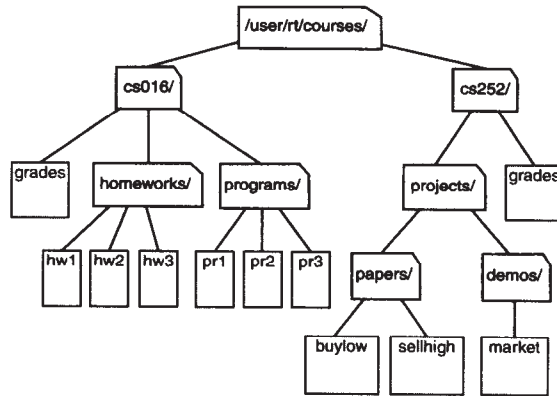


图2-8 表示部分文件系统的树

75

示例2.2 在大多数操作系统中，文件被分层组织到嵌套目录 (也称文件夹) 中，为用户呈现出的是树形结构，如图2-8所示。更确切地讲，树的内部结点与目录关联，外部结点与常规文件关联。在UNIX/Linux操作系统中，树的根相应地称为“根目录”，用符号“/.”表示。它是UNIX/Linux文件系统中所有目录和文件的祖先。

如果对于每个结点的子结点存在一个定义的线性顺序，则称树是有序的 (ordered)，即可以将一个结点的子结点按序排列。有序树表明在它的兄弟结点之间存在线性关系，即可以用正确的顺序将它们表成线性序列或迭代器。

示例2.3 结构化文档 (如图书) 可以分层组织成一棵树，其内部结点是章、节和子节，外部结点是段落、表、图和参考书目等。如图2-9所示。还可以进一步扩展树，使段落由句子组成，句子由单词组成，单词由字符组成。在任何情况下，这样一棵树都是有序树的一个例子，因为在每个结点的子结点之间存在良好定义的次序。

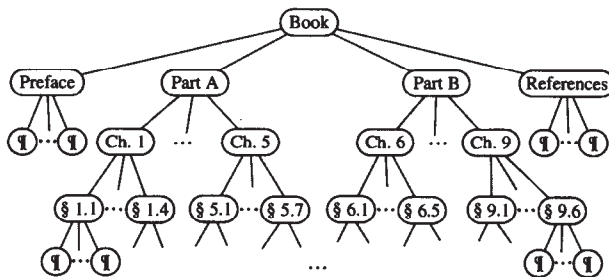


图2-9 一本书关联的一棵树

二叉树是一棵有序树，其中每个结点至多有两个子结点。如果每个内部结点有两个子结点，则称二叉树是真 (proper) 二叉树。对于二叉树中的每个内部结点，可以把每个子结点标注为左子结点 (left child) 或右子结点 (right child)。这些子结点是有序的，左子结点在右子结点之前。

根在内部结点 v 的左子结点/右子结点上的子树称为 v 的左子树(left subtree)/右子树(right subtree)。本书约定,除非特别声明,否则每棵二叉树都是一棵真二叉树。当然,即使对于一棵非真二叉树,它仍然是一般意义上的树,每个结点至多有两个子结点。二叉树应用广泛。

76

示例2.4 可用一棵树表示算术表达式,该树的外部结点关联变量或常量,它的内部结点关联某个运算符 $+$ 、 $-$ 、 \times 和 $/$ 。如图2-10所示。这样一棵树中的每个结点都有一个与其关联的值。

- 如果一个结点是外部结点,那么它的值就是变量或常量的值。
- 如果一个结点是内部结点,那么它的值则通过对其子结点的值应用各种运算来定义。

这样一棵算术表达式树是一棵真二叉树,因为它的每个运算符 $+$ 、 $-$ 、 \times 和 $/$ 只取两个操作数。当然,如果允许使用一元运算符,像 $-x$ 中的负号 $(-)$,那么可得到一棵非真二叉树。

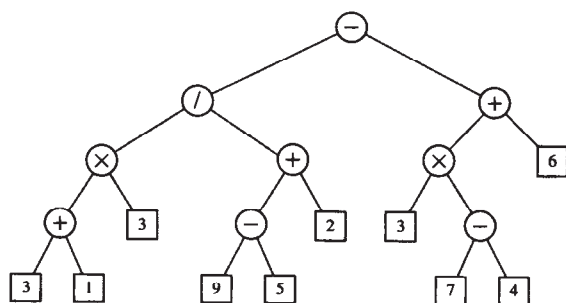


图2-10 表示算术表达式的二叉树。这棵树表示表达式 $((((3+1)\times 3)/((9-5)+2))-(3\times(7-4))+6))$ 。以“/”标记的内部结点关联的值为2

2.3.1 树抽象数据类型

树ADT将元素存储在位置中。正如表中的位置那样,这里的位置是相对于近邻位置定义的。树中的位置(position)是它的结点(node),近邻位置满足定义了一棵有效树的父-子关系。因此,对于树可以互换使用术语“位置”和“结点”。如同表中的位置一样,树中的位置对象支持element()方法,它返回该位置的对象。树中结点位置的实际能力来自于树ADT的如下访问方法(accessor method):

- root(): 返回树的根。
- parent(v): 返回结点 v 的父结点;如果 v 是根结点,则出错。
- children(v): 返回结点 v 的子结点的一个迭代器。

如果树 T 是有序的,那么迭代器children(v)提供按序访问 v 的子结点。如果 v 是一个外部结点,那么children(v)是一个空迭代器。

77

此外,还包括如下查询方法(query method):

- isInternal(v): 测试结点 v 是否为内部结点。
- isExternal(v): 测试结点 v 是否为外部结点。
- isRoot(v): 测试结点 v 是否为根结点。

还有许多树应当支持的方法,它们未必与树结构有关。这样的泛型方法包括:

- size(): 返回树中的结点数。
- elements(): 返回存储在树的结点中所有元素的一个迭代器。
- positions(): 返回树的所有结点的一个迭代器。
- swapElements(v, w): 交换存储在结点 v 和 w 中的元素。

- `replaceElement(v, e)`: 用 e 替换存储在结点 v 中的元素, 并返回原结点 v 中的元素。

这里没有定义树的任何特殊更新方法。而是留待以后与特定树应用一起定义不同的树更新方法。

2.3.2 树的遍历

本节介绍在树上进行计算的一些算法, 它们通过树ADT的方法访问这棵树。

1. 假设

为了分析基于树的算法的运行时间, 对于树ADT方法的运行时间做出以下假设。

- 访问方法`root()`和`parent(v)`的运行时间为 $O(1)$ 。
- 查询方法`isInternal(v)`、`isExternal(v)`和`isRoot(v)`的运行时间也为 $O(1)$ 。
- 访问方法`children(v)`的运行时间为 $O(c_v)$, 其中 c_v 是 v 的子结点数。
- 泛型方法`swapElements(v, w)`和`replaceElement(v, e)`的运行时间为 $O(1)$ 。
- 返回迭代器的泛型方法`elements()`和`positions()`的运行时间为 $O(n)$, 其中 n 为树中的结点数。
- 对于由方法`elements()`、`positions()`和`children(v)`以及方法`hasNext()`、`nextObject()`或`nextPosition()`返回的迭代器, 它们每一个所需要的时间都为 $O(1)$ 。

78

在2.3.4节中介绍的树数据结构满足上述假设。不过, 在描述用具体数据结构如何实现树ADT之前, 先来描述树ADT的一些方法, 这些方法可用来解决树的一些有趣问题。

2. 深度和高度

设 v 是树 T 的一个结点, v 的深度(depth)就是 v 的祖先数(不含 v 自身)。注意这个定义蕴涵着树 T 的根的深度为0。结点 v 的深度也可递归定义如下:

- 如果 v 是根, 则 v 的深度为0。
- 否则, v 的深度为1加上 v 的父结点的深度。

基于上述定义, 算法2-6所示的递归算法`depth`通过对 v 的父结点递归地调用自己, 再加上1作为返回值, 来计算 T 中结点 v 的深度。

算法2-6 计算树 T 中结点 v 的深度的算法`depth`

```

算法 depth( $T, v$ ):
    if  $T.isRoot(v)$  then
        return 0
    else
        return 1 + depth( $T, T.parent(v)$ )

```

算法`depth(T, v)`的运行时间为 $O(1+d_v)$, 其中 d_v 表示结点 v 在树 T 中的深度, 这是由于对于 v 的每个祖先结点, 算法都会执行常数递归步骤。因此, 在最坏情况下, `depth`算法运行时间为 $O(n)$, 其中 n 为树 T 中的结点总数。尽管运行时间是输入大小的函数, 还是可以根据参数 d_v 更精确地表征运行时间, 因为这将会比 n 小得多。

树 T 的高度(height)等于 T 的一个外部结点的最大深度。虽然这个定义是正确的, 但它不能导致一个有效算法。的确, 如果将上述求深度的算法应用于树 T 中的每个结点, 就会导出计算 T 的高度所需时间为 $O(n^2)$ 。利用关于树 T 中结点 v 的高度的以下递归定义, 就可导出更好的算法。

- 如果 v 是一个外部结点, 则 v 的高度为0。
- 否则, v 的高度为1加上 v 的子结点的最大高度。

79

树 T 的高度(height)就是树 T 中根的高度。

算法2-7所示的算法height利用上述关于高度的递归定义,有效地计算树 T 的高度。递归方法height(T, v)表示该算法计算根为 v 的树 T 的子树的高度。树 T 的高度通过调用height($T, T.root()$)而得。

算法2-7 计算以 v 为根的树 T 的子树高度的算法height

```

算法 height( $T, v$ ):
    if  $T.isExternal(v)$  then
        return 0
    else
         $h \leftarrow 0^1$ 
        while each  $w \in T.children(v)$  do
             $h \leftarrow \max(h, height(T, w))^2$ 
        return  $1 + h$ 

```

height算法是递归算法,如果起初在树 T 的根部调用它,则最终会在 T 的每个结点上调用它一次。因此,可以用平摊方法论述这个方法的运行时间。首先确定每个结点(在非递归部分)所花费的时间,然后将所有结点所需的时间求和。迭代器children(v)的计算所需时间为 $O(c_v)$,其中 c_v 是结点 v 的子结点数。此外,for循环迭代 c_v 次,每次迭代所需时间为 $O(1)$ 加上递归调用 v 的子结点的时间。因此,算法height在每个结点上所花的时间为 $O(1 + c_v)$,它的运行时间为 $O\left(\sum_{v \in T} (1 + c_v)\right)$ 。为了完成分析,利用以下性质。

定理2.1 设 T 是具有 n 个结点的一棵树, c_v 是 T 中结点 v 的子结点数。那么,

$$\sum_{v \in T} c_v = n - 1$$

证明 T 的每个结点(除根结点外)都是另一个结点的子结点,因此每个结点对于求和 $\sum_{v \in T} c_v$ 贡献一个单位。 ■

由定理2.1可知,当在 T 的根上调用时,算法height的运行时间为 $O(n)$,其中 n 为树 T 中的结点数。

树 T 的遍历(traversal)是对树 T 中的所有结点进行的一种系统访问或“拜访”。下面介绍的树的基本遍历模式称为前序遍历和后序遍历。

80

3. 前序遍历

在树 T 的前序(preorder)遍历中,首先访问 T 的根,然后递归遍历以其子结点为根的子树。如果树是有序的,那么子树遍历的顺序则遵循子结点的顺序。与访问结点 v 关联的特定行为取决于这个遍历的应用,它可以是递增计数器,也可以是对 v 进行某些复杂计算。算法2-8显示了前序遍历根为 v 的子树的伪代码。初始调用例程为preorder($T, T.root()$)。

算法2-8 前序遍历算法preorder

```

算法 preorder( $T, v$ ):
    执行结点 $v$ 的“访问”行为
    for 对于 $v$ 的每个子结点 $w$  do
        通过调用preorder( $T, w$ ),递归遍历以 $w$ 为根的子树

```

1. 原书为 $h = 0$ 。——译者注
2. 原书为 $h = \max(h, height(T, w))$ 。——译者注

前序遍历算法用于产生树中结点的一个线性序列,在这个线性序列中,父结点总是在其子结点之前。这样的序列有多种不同的应用。下面的例子给出了这种应用的一个简单实例。

示例2.5 与文档有关的树的前序遍历(如示例2.3所示)从头到尾顺序检查整个文档。如果在遍历之前删除外部结点,那么遍历检查文档的目录表。如图2-11所示。

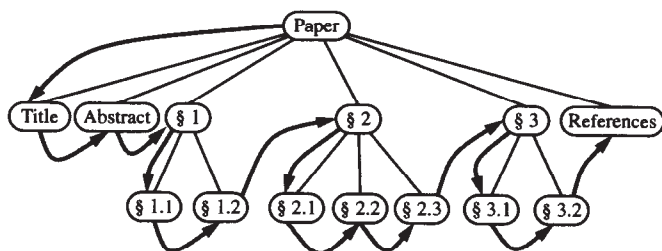


图2-11 有序树的前序遍历

对前序遍历的分析实际上类似于上面对height算法所做的分析。在每个结点 v 处,前序遍历算法的非递归部分所需时间为 $O(1 + c_v)$,其中 c_v 是结点 v 的子结点数。因此,由定理2.1可知,前序遍历树 T 的总运行时间为 $O(n)$ 。

81

4. 后序遍历

后序(postorder)遍历是另一种重要的树遍历算法。这个算法可被看成前序遍历的逆过程。因为它首先递归地遍历以根的子结点为根的子树,然后访问根。但是它类似于前序遍历,这在于可以通过特例化与“访问”结点 v 关联的某个动作,来解决特定的问题。如同前序遍历,如果树是有序的,则可以按照它们指定的顺序,对结点 v 的子结点进行递归调用。后序遍历的伪代码如算法2-9所示。

算法2-9 后序遍历算法postorder

```

算法 postorder( $T, v$ ):
  for 对于 $v$ 的每个子结点 $w$  do
    通过调用postorder( $T, w$ ), 递归遍历以 $w$ 为根的子树
  访问结点 $v$ 
  
```

后序遍历这个名称来自于这样一个事实,即该遍历方法首先访问以结点 v 为根的子树中的所有其他结点,然后访问结点 v 。如图2-12所示。

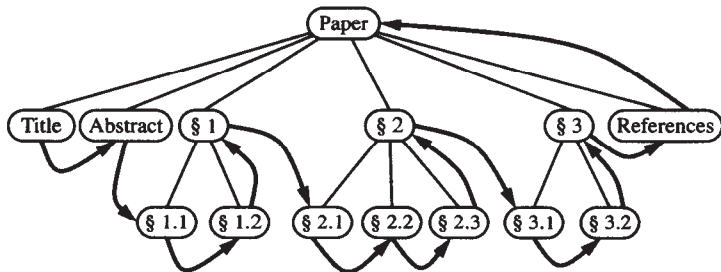


图2-12 图2-11的有序树的后序遍历

后序遍历的运行时间分析类似于对前序遍历所做的分析。算法中非递归部分所花费的总时间与花费在访问树中每个结点的子结点上的时间成正比。因此,对具有 n 个结点的树进行后序

遍历所需要的时间为 $O(n)$ ，假设访问每个结点所需的时间为 $O(1)$ ，即后序遍历算法的运行时间是线性的。

后序遍历方法可用于解决某些问题，这些问题中，希望计算树中每个结点 v 的某种性质，但是计算结点 v 的那种性质要求已经计算了 v 的子结点的同一性质。以下例子说明了这种应用。

82

示例2.6 考虑文件系统的树 T ，其中外部结点表示文件，内部结点表示目录（习题2.2）。假定想要计算目录所用的磁盘空间，它可通过对下列各项求和递归地给出：

- 目录自身的大小。
- 目录中的文件大小。
- 子目录所用的空间。

如图2-13所示。可通过后序遍历树 T 计算目录所用的磁盘空间。在遍历内部结点 v 的子树之后，通过将目录 v 自身的大小、包含在目录 v 中的文件大小以及 v 的每个内部子结点所用的空间大小（这是通过递归地后序遍历 v 的子结点计算得到的）这几项相加，计算 v 所用的空间。

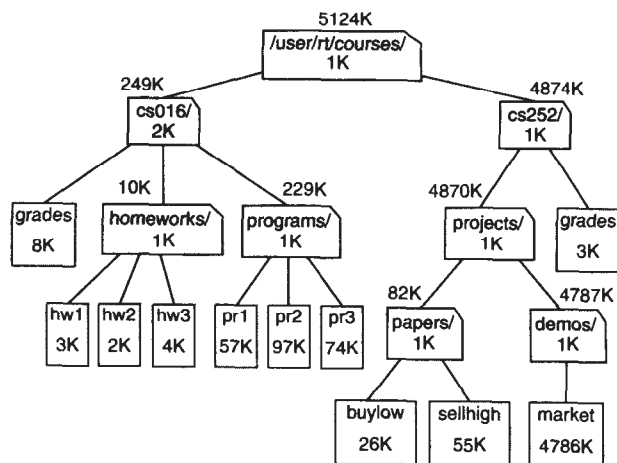


图2-13 表示图2-8中的文件系统的树，其中显示了每个结点内部关联的文件/目录的名称和大小，以及每个内部结点上方的关联目录所用的磁盘空间

尽管前序遍历和后序遍历是访问树中结点的常用方法。也可设想还有其他遍历方法。例如，可以遍历一棵树，以便先访问树中深度为 $d+1$ 的结点，再访问树中深度为 d 的所有结点。例如，可以利用一个队列实现这样的遍历，而前序遍历和后序遍历则利用栈（在利用递归描述这些方法时，这个栈是隐含的；但也可以显式地利用栈，以避免使用递归）。此外，下面将讨论的二叉树支持另一种遍历方法，称之为中序遍历。

83

2.3.3 二叉树

二叉树是一种特别令人感兴趣的树。正如在2.3节中所提到，真二叉树（binary tree）是一棵有序树，其中每个内部结点恰好都有两个孩子。本书约定，除非特别说明，否则假设二叉树是真二叉树。注意，针对于二叉树的约定不失一般性。正如在研究习题C-2.14时所做的那样，可以轻松地把任何非真二叉树转换成真二叉树。即使没有这种约定，也可把一棵非真二叉树视为真二叉树，这只需简单地把遗失的外部结点看作“空结点”或者仍会被统计为结点的占位符即可。

1. 二叉树抽象数据类型

作为一种抽象数据类型，二叉树是树的一种特殊形式。它支持另外三个访问方法：

- `leftChild(v)`: 返回 v 的左子结点；如果 v 是一个外部结点，则出错。
- `rightChild(v)`: 返回 v 的右子结点；如果 v 是一个外部结点，则出错。
- `sibling(v)`: 返回结点 v 的兄弟结点；如果 v 是根结点，则出错。

注意，如果将这些方法应用到非真二叉树，一定还有其他的错误条件。例如，在一棵非真二叉树中，一个内部结点可能没有左子结点或右子结点。这里没有给出更新二叉树的方法，因为这些方法可根据特定应用的需要而建立。

2. 二叉树的性质

把树 T 中具有相同深度 d 的所有结点的集合定义为 T 的层 (level) d 。在二叉树 T 中，层0只有一个结点（根结点），层1至多有两个结点（根的子结点），层2至多有4个结点，依此类推。如图2-14所示。一般而言，层 d 至多有 2^d 个结点。这蕴涵着以下定理（证明留作习题R-2.4）。

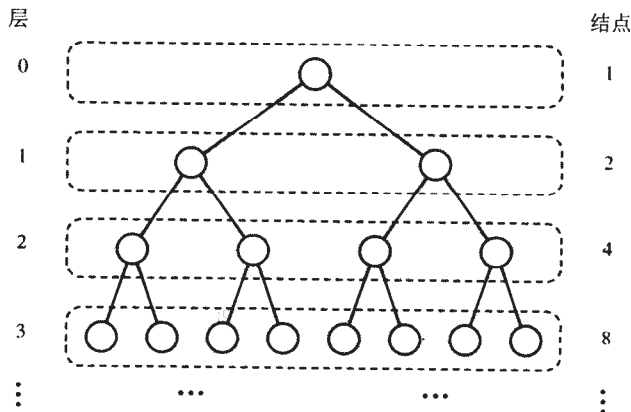


图2-14 二叉树各层中的最大结点数

定理2.2 设 T 是一棵具有 n 个结点的（真）二叉树， h 表示 T 的高度。那么， T 具有如下性质：

- (1) T 中的外部结点数至少为 $h+1$ ，至多为 2^h 。
- (2) T 中的内部结点数至少为 h ，至多为 2^h-1 。
- (3) T 中的结点总数至少为 $2h+1$ ，至多为 $2^{h+1}-1$ 。
- (4) T 的高度至少为 $\log(n+1)-1$ ，至多为 $(n-1)/2$ ，即 $\log(n+1)-1 \leq h \leq (n-1)/2$ 。

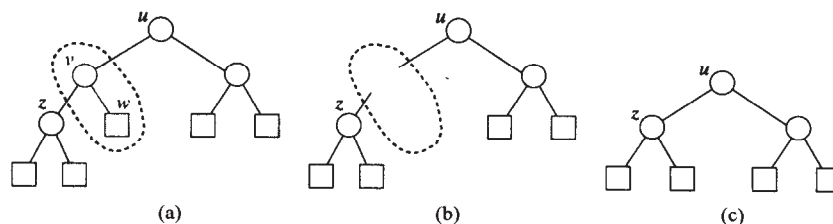
84

定理2.3 在一棵（真）二叉树中，外部结点数比内部结点数多1个。

证明 用归纳法证明。如果 T 只有一个结点 v ，那么 v 是外部结点，命题显然成立。否则，从 T 中删除任意一个外部结点 w 及其父结点 v （ v 是一个内部结点）。如果 v 有父结点 u ，那么重新连接 u 与 w 以前的兄弟结点 z ，如图2-15所示。称这个操作为`removeAboveExternal(w)`，它删除一个内部结点和一个外部结点，并使树 T 变为一棵真二叉树。因此，由归纳假设，这棵真二叉树中的外部结点数要比内部结点数多1。因为删除了一个内部结点和一个外部结点，使树 T 成为这棵较小的树，因此，同样性质必然对 T 成立。 ■

一般而言，对于非二叉树，上述关系不能成立。

在后面的节章中，将讨论上述事实的一些重要应用。不过在讨论这些应用之前，首先应当了解有关如何遍历和表示二叉树的更多知识。

图2-15 removeAboveExternal(w)操作。它删除一个外部结点及其父结点，在定理2.3的证明中使用

85

3. 遍历二叉树

像一般的树一样，在二叉树上进行的计算常常与树的遍历有关。在本节中将介绍利用二叉树ADT的方法表示二叉树的遍历算法。关于它的运行时间，除了2.3.2节中所做的对树ADT方法的运行时间的假设之外，还假设二叉树的方法children(v)所需的时间为 $O(1)$ ，这是因为每个结点有0个或两个子结点。同样，假设方法leftChild(v)、rightChild(v)和sibling(v)所需的时间为 $O(1)$ 。

4. 前序遍历二叉树

因二叉树也可看作一棵一般的树，一般树的前序遍历（算法2-8）可用于任何树。但是，对于二叉树的遍历，可以简化伪代码，如算法2-10所示。

算法2-10 算法binaryPreorder对以 v 为根的二叉树 T 的子树进行前序遍历

```

算法 binaryPreorder( $T, v$ ):
    访问结点 $v$ 
    if  $v$ 是一个内部结点 then
        binaryPreorder( $T, T.\text{leftChild}(v)$ )    {递归遍历左子树}
        binaryPreorder( $T, T.\text{rightChild}(v)$ )    {递归遍历右子树}

```

5. 后序遍历二叉树

类似地，一般树的后序遍历（算法2-9）也可特别地用于二叉树，如算法2-11所示。

算法2-11 算法binaryPostorder对以 v 为根的二叉树 T 的子树进行后序遍历

```

算法 binaryPostorder( $T, v$ ):
    if  $v$ 是一个内部结点 then
        binaryPostorder( $T, T.\text{leftChild}(v)$ )    {递归遍历左子树}
        binaryPostorder( $T, T.\text{rightChild}(v)$ )    {递归遍历右子树}
    访问结点 $v$ 

```

有趣的是，针对一般树的前序遍历方法和后序遍历方法可特别用于二叉树，这提出了遍历二叉树的第三种方法，这种方法不同于前序遍历和后序遍历方法。

86

6. 中序遍历二叉树

中序遍历是另一种遍历二叉树的方法。在这种遍历中，访问一个位于其左、右子树的递归遍历之间的结点，如算法2-12所示。

算法2-12 算法inorder对以 v 为根的二叉树 T 的子树进行中序遍历

```

算法 inorder( $T, v$ ):
    if  $v$ 是一个内部结点 then
        inorder( $T, T.\text{leftChild}(v)$ )    {递归遍历左子树}
    访问结点 $v$ 
    if  $v$ 是一个内部结点 then
        inorder( $T, T.\text{rightChild}(v)$ )    {递归遍历右子树}

```


可将二叉树 T 的中序遍历看作“从左到右”访问 T 的结点。实际上，对于每个结点 v ，中序遍历在访问 v 的左子树中的所有结点之后和访问 v 的右子树中的所有结点之前，访问 v 结点，如图2-16所示。

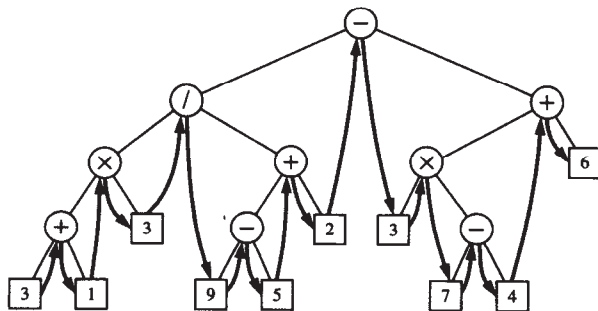


图2-16 中序遍历二叉树

7. 统一的树遍历框架

到目前为止讨论的树遍历算法是各种形式的迭代器。每种遍历算法按照一定的顺序访问树中的结点，而且保证树中的每个结点都正好会被访问一次。但是，如果放松对树中的每个结点正好访问一次的条件，可将上面给出的树遍历算法统一到单一设计模式中。得到的遍历方法称为欧拉路径遍历法（Euler tour traversal），下面学习这种方法。这种遍历方法的优点是允许轻松表达各种更一般的算法。

87

8. 欧拉路径遍历二叉树

可把欧拉路径遍历一棵二叉树 T 非形式定义为围绕 T 的“遍历”。从根结点开始，向着它的左子结点，将 T 的边看作总是保持在我们左边的“墙”。如图2-17所示。对于 T 中的每个结点 v ，欧拉路径都会三次遇到它们：

- “在左边”（在 v 的左子树的欧拉路径之前）。
- “从下面”（介于 v 的两棵子树的欧拉路径之间）。
- “在右边”（在 v 的右子树的欧拉路径之后）。

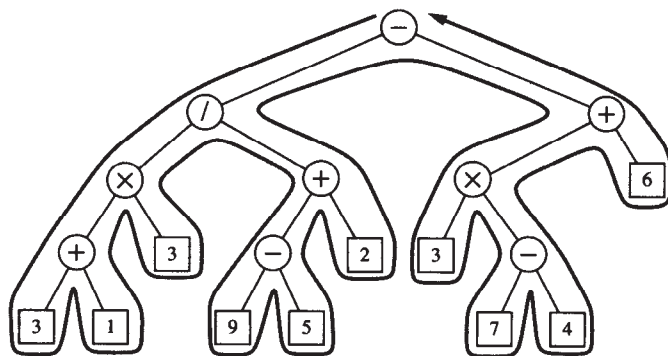


图2-17 欧拉路径遍历二叉树

如果 v 是外部结点，那么实际上这三个“访问”会同时发生。

算法2-13给出了以结点 v 为根的子树的欧拉路径的伪代码。

算法2-13 算法eulerTour对以 v 为根的二叉树 T 的子树进行欧拉路径遍历

算法 eulerTour(T, v):

在左边执行访问结点 v 的动作

if v 是一个内部结点 **then**

 调用eulerTour($T, T.\text{leftChild}(v)$), 递归遍历 v 的左子树

从下面执行访问结点 v 的动作

if v 是一个内部结点 **then**

 调用eulerTour($T, T.\text{rightChild}(v)$), 递归遍历 v 的右子树

在右边执行访问结点 v 的动作

如果仅当在左边相遇时, 每个结点才具有一个关联的“访问”动作, 则二叉树的前序遍历等价于欧拉路径遍历。同样, 如果仅当分别从下面和在右边相遇时, 每个结点才具有一个关联的“访问”动作, 则二叉树的中序遍历和后序遍历等价于欧拉路径遍历。

88

欧拉路径遍历扩展了前序遍历、中序遍历和后序遍历方法, 但它还可进行其他一些遍历。例如, 假定在一个有 n 个结点的树 T 中, 想要计算每个结点 v 的后代数。首先将计数器初始化为0, 开始欧拉路径计数, 然后每当访问一个左边的结点时, 使计数器增1。为了确定结点 v 的后代数, 计算当 v 在左边被访问和在右边被访问时, 这两个计数器之间的差值, 并且加1作为 v 的后代数。这个简单的规则给出了 v 的后代数, 因为在左边访问 v 和在右边访问 v 之间, 会对根为 v 的子树中的每个结点进行计数。因此, 计算 T 中每个结点的后代数量所需时间为 $O(n)$ 。

假设访问一个结点所需时间为 $O(1)$, 容易分析欧拉路径遍历的运行时间。也就是说, 在每次遍历中, 树的每个结点在遍历期间花费的时间为常数, 因此 n 个结点树的总运行时间为 $O(n)$ 。

对于一个算术表达式, 欧拉路径遍历的另一种应用是从该表达式的表达式树(示例2.4)输出一个完全加括号的算术表达式。算法2-14中所示的方法printExpression通过在欧拉路径中执行以下操作, 来完成这一任务。

- “在左边”动作: 如果结点是内部结点, 输出“(”。
- “从下面”动作: 输出存储在结点中的值或运算符。
- “在右边”动作: 如果结点是内部结点, 输出“)”。

算法2-14 输出算术表达式的算法, 该算术表达式与以 v 为根的算术表达式树 T 的子树相关联

算法 printExpression(T, v):

if $T.\text{is External}(v)$ **then**

 输出存储在 v 中的值

else

 输出“(”

 printExpression($T, T.\text{leftChild}(v)$)

 输出存储在 v 中的运算符

 printExpression($T, T.\text{rightChild}(v)$)

 输出“)”

给出了这些伪代码的例子之后, 现在描述用具体数据结构(如序列和链表结构)实现树抽象数据类型的许多有效的方法。

89

2.3.4 表示树的数据结构

本节描述表示树的具体数据结构。

1. 用基于向量的结构表示二叉树

表示二叉树 T 的简单结构基于对 T 中结点的编号方式。对于树 T 中的每个结点 v ，设 $p(v)$ 表示整数，定义如下。

- 如果 v 是 T 的根，那么 $p(v) = 1$ 。
- 如果 v 是结点 u 的左子结点，那么 $p(v) = 2p(u)$ 。
- 如果 v 是结点 u 的右子结点，那么 $p(v) = 2p(u) + 1$ 。

编号函数 p 称为二叉树 T 中结点的层序编号 (level numbering)，因为它从左到右按增序方式对 T 中每一层的结点编号，尽管它可能跳过某些结点。如图2-18所示。

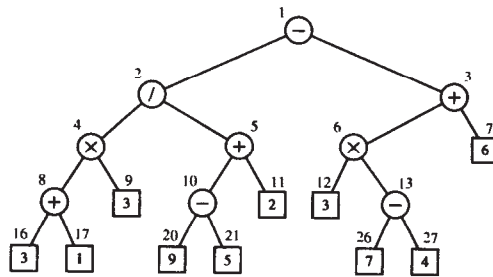


图2-18 二叉树层序编号示例

通过向量 S ，层序编号函数 p 给出了一种二叉树 T 的表示，其中 T 的结点 v 与 S 中位序为 $p(v)$ 的元素相关联（如图2-19所示）。一般可用一个可扩展数组实现向量 S （见1.5.2节）。这样的实现简单、快速，因为利用了对于数 $p(v)$ （它与操作中涉及的每个结点 v 相关联）的简单算术操作，很容易用这一实现执行方法`root`、`parent`、`leftChild`、`rightChild`、`sibling`、`isInternal`、`isExternal`和`isRoot`。也就是说，每个位置对象 v 只是向量 S 的索引 $p(v)$ 的“包装器”，我们把这种实现的细节留作一个简单的习题（R-2.7）。

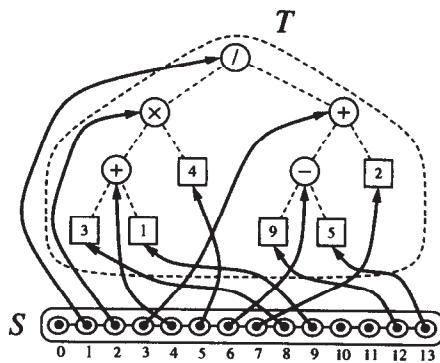


图2-19 通过向量 S 表示二叉树 T

设 n 是 T 中的结点数， p_M 是 T 中所有结点上的 $p(v)$ 的最大值。因为 S 中位序为0的元素与 T 中的任何结点无关，向量 S 的大小为 $N = p_M + 1$ 。同时，一般来讲，向量 S 中有大量空元素，它们没有引用 T 中现有的结点。例如，这些空位置可对应空外部结点，或甚至对应这些结点的后代可能去的地

方。事实上,在最坏情况下, $N = 2^{(n+1)/2}$, 其证明留作习题 (R-2.6)。在2.4.3节中,将看到一类称为“堆”的二叉树,有 $N = n + 1$ 。此外,如果所有外部结点为空,正如堆实现中那样,那么不对向量 S 的大小进行扩展,则不包括其下标是树中最后一个内部结点过去的下标的那些外部结点,从而节省额外的空间。因此,尽管有最坏情况下的使用空间,但对于某些应用,二叉树的向量表示在空间上仍是有效的。尽管如此,对于一般的二叉树,这种表示的指数级最坏情况下需要的指数空间是相当昂贵的。

表2-3总结了向量实现的二叉树方法的运行时间。表中没有包括用于更新二叉树的任何方法。

表2-3 用向量 S 实现的二叉树 T 的方法的运行时间,其中 S 用数组实现。用 n 表示 T 中的结点数,用 N 表示 S 的大小。迭代器 `elements()`、`positions()` 和 `children(v)` 中的方法 `hasNext()`、`nextObject()` 和 `nextPosition()` 所需的时间为 $O(1)$, 所需的空间为 $O(N)$ 。在最坏情况下,所需的空间为 $O(2^{(n+1)/2})$

操 作	时 间
<code>positions</code> 、 <code>elements</code>	$O(n)$
<code>swapElements</code> 、 <code>replaceElement</code>	$O(1)$
<code>root</code> 、 <code>parent</code> 、 <code>children</code>	$O(1)$
<code>leftChild</code> 、 <code>rightChild</code> 、 <code>sibling</code>	$O(1)$
<code>isInternal</code> 、 <code>isExternal</code> 、 <code>isRoot</code>	$O(1)$

二叉树的向量实现快速、简单,但如果树的高度很大,则空间利用率就非常低。以下讨论的用于表示二叉树的数据结构避免了这个缺点。

2. 用链表结构表示二叉树

一种实现二叉树 T 的自然方式是采用链表结构 (linked structure)。在这种方法中,用一个对象表示 T 中的每个结点,该对象引用存储在 v 中的元素以及与 v 的子结点和父结点关联的位置。图2-20显示了二叉树的链表结构表示。

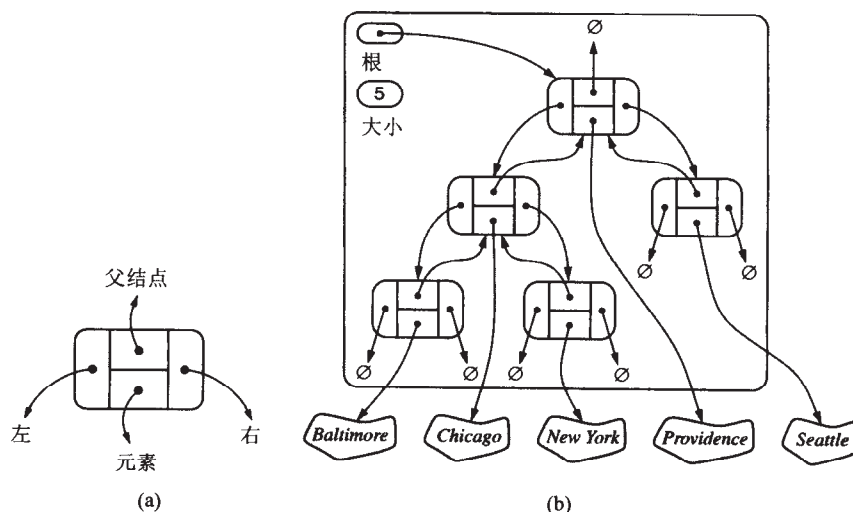


图2-20 链表数据结构表示二叉树示例: (a)关联结点的对象; (b)具有5个结点的二叉树的完整数据结构

如果 v 是 T 的根,那么对 v 父结点的引用为 `null`; 如果 v 是外部结点,那么对 v 子结点的引用为 `null`。

对于外部结点为空时的情况,如果想要节省空间,那么可令对外部空结点的引用为null,即允许来自内部结点的对一个外部结点的子结点的引用为null。

此外,可用 $O(1)$ 时间直接实现方法size()、isEmpty()、swapElements(v, w)和replaceElement(v, e)。此外,进行一次中序遍历就能实现方法positions(),方法elements()的实现是类似的。因此,方法positions()和方法elements()每个都需要 $O(n)$ 的时间。

考虑这种数据结构所需的空间,注意对于树 T 的每个结点,存在一个常量大小的对象。因此,所需的总空间为 $O(n)$ 。

3. 用链表结构表示一般树

对表示二叉树的链表结构进行扩充,使其能表示一般树。因为一般树中的结点 v 所能具有的子结点数没有限制,利用容器(例如表或向量)而不是利用实例变量存储 v 的子结点。这种结构的系统说明如图2-21所示。假设把结点的容器实现为一个序列。

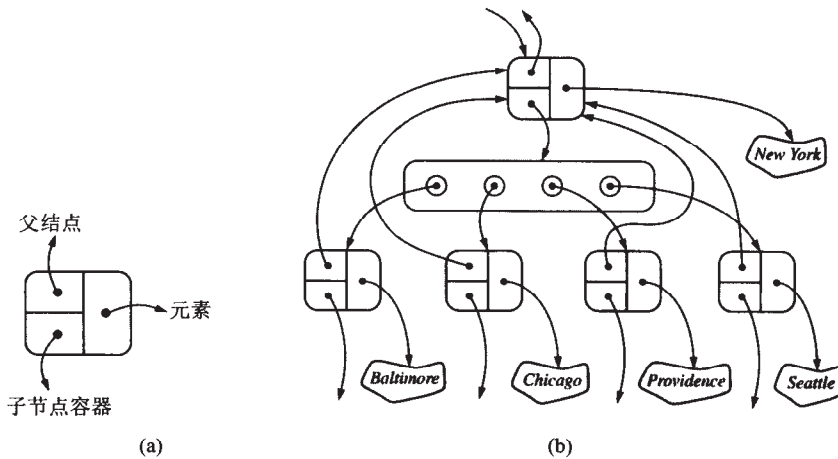


图2-21 用链表结构表示树: (a)对象关联一个结点; (b)关联一个结点及其子结点的数据结构部分

注意树ADT的链表实现的性能(如表2-4所示)类似于二叉树的链表实现的性能,其主要区别在于,在树ADT的实现中,使用了一个高效的容器(如表或向量)存储每个结点 v 的子结点,而不是直接链接到正好两个子结点上。

表2-4 用链表结构实现的 n 个结点树的方法的运行时间。设 c_v 表示结点 v 的子结点数

操 作	时 间
size、isEmpty	$O(1)$
positions、elements	$O(n)$
swapElements、replaceElement	$O(1)$
root、parent	$O(1)$
children(v)	$O(c_v)$
isInternal、isExternal、isRoot	$O(1)$

2.4 优先队列和堆

本节提供基于关键字和比较器的概念研究优先队列的一个框架。

2.4.1 优先队列抽象数据类型

许多应用常常要求按照参数或某些性质，对对象进行比较和排列。这些参数或性质称为“关键字”。集合中的每个对象都被赋予一个关键字。关键字（key）的形式定义为一个对象，它被赋予一个元素，作为该元素的特定属性，可用于标识、排列和衡量元素。注意关键字通常由用户或应用赋予一个元素。

因此，关键字作为任一对象类型的概念具有一般意义。为使关键字的这种一般定义具有一致性，并且仍然能够讨论什么时候一个关键字的优先级比另一个大，需要稳健地定义比较关键字的一条规则。也就是说，优先队列需要比较规则，这个规则决不会与自身发生矛盾。为了使这个比较规则（用 \leq 符号表示）在这方面是稳健的，必须定义一个全序（total order）关系，即为每对关键字之间都定义了比较规则，并且它必须满足以下性质：

- 自反性： $k \leq k$ 。
- 反对称性：如果 $k_1 \leq k_2$ 且 $k_2 \leq k_1$ ，那么 $k_1 = k_2$ 。
- 传递性：如果 $k_1 \leq k_2$ 且 $k_2 \leq k_3$ ，那么 $k_1 \leq k_3$ 。

任何满足这三个性质的比较规则 \leq 永不会导致比较矛盾。事实上，这个规则定义了一组关键字之间的一个线性排序关系。因此，如果元素的有限集定义了一个全序关系，那么就明确地定义了最小（smallest）关键字 k_{min} 的概念，即对于集合中任一关键字 k ，都有 $k_{min} \leq k$ 。

优先队列（priority queue）是元素的一个容器，每个元素都有一个关键字，它是在插入元素时提供的。“优先队列”这个名称的来由是，关键字确定优先级，可用优先级选出要被移除的元素。优先队列 P 的两个基本方法如下。

- `insertItem(k, e)`：将关键字为 k 的元素 e 插入 P 。
- `removeMin()`：从 P 中删除并返回具有最小关键字的元素，即这个元素的关键字小于或等于 P 中其他任何元素的关键字。

顺便说一句，称`removeMin`方法为`extractMin`方法，以便强调这个方法删除并同时返回 P 中的最小元素。可以扩充这两个方法，支持方法有`size()`和`isEmpty()`。同时，也可以添加如下访问方法：

- `minElement()`：返回（但不删除） P 中具有最小关键字的元素。
- `minKey()`：返回（但不删除） P 中的最小关键字。

如果优先队列为空，这两个方法都返回错误条件。

优先队列ADT的有趣方面之一是，它比序列ADT更简单，这一点现在应当很明显。这种简洁性是由于，优先队列中元素的插入和删除完全基于它们的关键字。而序列中元素的插入和删除则基于位置和位序。

比较器

优先队列ADT隐含着利用了软件工程的设计模式即比较器。这种模式确定了比较关键字的方式，它设计成支持大多数常规和可重用的优先队列。对于这样的设计，不应该依靠关键字提供它们的比较规则，因为这样的规则可能不是用户想要的（尤其是对于多维数据）。作为替代，利用特殊比较器（comparator）对象，可从外部为关键字提供比较规则。比较器是一个比较两个关键字的对象。假设在构造优先队列 P 时，给定 P 的一个比较器，并且还可以设想如果一个优先队列的旧比较器已经“过时”，则会提供给优先队列新比较器的能力。当 P 需要比较两个关键字时，利用给定的比较器进行这种比较。因此，程序员可以编写一个通用优先队列实现，它在广泛的环境中都可正确工作。形式上，一个比较器对象提供如下方法，每个方法取两个关键字进行比较（如果关键字不能进行比较，则报告出错）。比较器ADT包括如下方法：

95

- $\text{isLess}(a, b)$: 当且仅当 a 小于 b 时, 为真。
- $\text{isLessOrEqualTo}(a, b)$: 当且仅当 a 小于或等于 b 时, 为真。
- $\text{isEqualTo}(a, b)$: 当且仅当 a 等于 b 时, 为真。
- $\text{isGreater}(a, b)$: 当且仅当 a 大于 b 时, 为真。
- $\text{isGreaterOrEqualTo}(a, b)$: 当且仅当 a 大于或等于 b 时, 为真。
- $\text{isComparable}(a)$: 当且仅当 a 可比较时, 为真。

2.4.2 PQ 排序、选择排序和插入排序

本节讨论如何利用优先队列对元素集进行排序。

1. PQ排序: 利用优先队列排序

在排序 (sorting) 问题中, 给定 n 个元素的集合 C , 可按照全序关系对它们进行比较。我们希望按增序重新排列它们 (如果存在tie, 则至少按非降序排列)。用优先队列 Q 对 C 进行排序的算法相当简单, 由以下两个阶段组成:

- (1) 在第一个阶段, 利用 n 个insertItem操作序列, 将 C 中的元素放入一个初始为空的优先队列 P 中, 对每个元素执行一次insertItem操作。
- (2) 在第二个阶段, 利用 n 个removeMin操作序列, 按非降序从 P 中提取元素, 并按顺序把它们放回 C 中。

算法的伪代码描述如算法2-15所示。假设 C 是一个队列 (例如表或向量)。算法对于任何优先队列 P 都能正确工作, 而与 P 的实现无关。然而, 算法的运行时间由操作insertItem和removeMin的运行时间决定, 而这两个方法的运行时间依赖于 P 的实现。实际上, 由于没有指定优先队列的实现方式, PriorityQueueSort应该更多考虑排序“模式”, 而不是考虑排序“算法”。PriorityQueueSort模式是几种流行的排序算法的范型, 包括选择排序、插入排序和堆排序, 它们将在本节余下部分讨论。

算法2-15 算法PQ-Sort。输入序列 C 中的元素同时作为优先队列 P 中的关键字和元素

算法 PQ-Sort(C, P):

输入: n 个元素序列 C 和优先队列 P , 利用全序关系比较作为 C 中元素的关键字

输出: 按全序关系排列的序列 C

while C 非空 **do**

$e \leftarrow C.\text{removeFirst}()$ {从 C 中删除元素 e }

$P.\text{insertItem}(e, e)$ {关键字为元素自身}

while P 非空 **do**

$e \leftarrow P.\text{removeMin}()$ {从 P 中删除最小的元素}

$C.\text{insertLast}(e)$ {将元素添加到 C 的末尾}

96

2. 使用基于无序序列实现的优先队列

作为优先队列 P 的第一种实现, 考虑对 P 中的元素及其在序列 S 中的关键字进行排序。假设 S 是一个数组或双向链表 (正如我们所见, 选择哪个实现将不会影响性能) 实现的常规序列。因此, S 中的元素为 (k, e) 对, 其中 e 是 P 中的元素, k 为它的关键字。实现 P 的方法insertItem(k, e)的一种简单方式是执行方法insertLast(p), 将新对象 $p = (k, e)$ 对添加到序列 S 的末尾。方法insertItem的实现所需时间为 $O(1)$, 这与序列是由数组还是链表实现的无关 (见2.2.3节)。这种选择意味着 S 可以是无序的, 因为总是将元素插入在 S 的末尾, 而不用考虑关键字的次序。因而, 要在 P 上执行操作minElement、minKey或者removeMin, 必须检查序列 S 中的所有元素, 才能找到 S 中具有最小 k 值

的元素 $p = (k, e)$ 。因此, 无论如何实现 S , P 上的这些查找方法所需时间都为 $O(n)$, 其中 n 为方法执行时 P 中元素个数。然而, 即使在最好情况下, 这些方法的运行时间也为 $\Omega(n)$, 因为每个方法都需查找整个序列才能找到最小的元素, 即这些方法的运行时间为 $\Theta(n)$ 。因此, 利用无序序列实现优先队列, 可获得常量时间的插入, 但`removeMin`需要线性时间。

3. 选择排序

如果用无序序列实现优先队列 P , 由于插入每个元素所需时间为常量时间, 那么PQ-Sort的第一阶段所需时间为 $O(n)$ 。在第二阶段中, 假设比较两个关键字所需时间为常量时间, 每个`removeMin`操作的运行时间与 P 中当前元素个数成正比。因此, 这个实现中的瓶颈计算是在第二阶段从一个无序序列中反复“选择”最小的元素。由于这个原因, 称这个算法为选择排序 (selection-sort)。

以下分析排序算法。如上所述, 瓶颈出在第二阶段, 它要反复从优先队列 P 中删除具有最小关键字的元素。 P 的大小开始为 n , 随着每个`removeMin`操作逐渐下降, 直到变为0。因此, 第一个`removeMin`操作所需时间为 $O(n)$, 第二个`removeMin`操作所需时间为 $O(n-1)$, 依此类推, 直到最后(第 n 个)操作所需时间为 $O(1)$ 。因此, 第二阶段所需的总时间为

$$O(n + (n-1) + \cdots + 2 + 1) = O\left(\sum_{i=1}^n i\right)$$

由定理1.3可知, $\sum_{i=1}^n i = n(n+1)/2$ 。因此, 第二阶段所需时间为 $O(n^2)$ 。这也是完整的选择排序算法的运行时间。

97

4. 使用基于有序序列实现的优先队列

另一种优先队列 P 的实现也是利用序列 S , 只不过元素按照关键字值有序存储。在这种情况下, 可以简单地用 S 中的`first`方法访问序列中的第一个元素, 实现方法`minElement`和`minKey`。同样, 也能把 P 的`removeMin`方法实现成`S.remove(S.first())`。假设 S 可用支持常量时间的, 前端元素删除(见2.2.3节)的数组或链表实现, 则查找并删除 P 中的最小元素所需时间为 $O(1)$ 。因此, 利用有序序列可以简单、快速地实现优先队列的访问和删除方法。

但是, 这种好处有一定的开销, 因为现在 P 的方法`insertItem`要求扫描整个序列 S , 才能找出插入新元素和关键字的合适位置。因此, 实现 P 的`insertItem`方法现在所需时间为 $O(n)$, 其中 n 为方法执行时 P 中的元素个数。总之, 利用有序序列实现优先队列时, 插入操作的运行时间为线性时间, 而查找和删除最小元素所需的时间为常量时间。

5. 插入排序

如果利用有序序列实现优先队列 P , 那么就能将PQ-Sort方法第二阶段的运行时间改进为 $O(n)$, P 上的每个`removeMin`操作所需时间为 $O(1)$ 。不幸的是, 第一阶段的运行时间现在变成了瓶颈。实际上, 在最坏情况下, 每次`insertItem`操作的运行时间与当前优先队列中的元素个数成正比。优先队列初始时元素个数为0, 其元素个数会不断增加, 直到 n 。第一个`insertItem`操作所需时间为 $O(1)$, 第二个`insertItem`操作所需时间为 $O(2)$, 依此类推。最后(第 n 个)`insertItem`操作最坏情况下所需时间为 $O(n)$ 。因此, 如果利用有序序列实现 P , 那么第一阶段变成瓶颈阶段。因此, 称这种排序算法为插入排序 (insertion-sort), 因为这个排序算法中的瓶颈在于反复将新元素“插入”到有序序列中的合适位置上。

分析插入排序的运行时间, 第一阶段在最坏情况下所需时间为 $O\left(\sum_{i=1}^n i\right)$ 。由定理1.3可知, 第一阶段运行时间为 $O(n^2)$, 则整个算法的运行时间为 $O(n^2)$ 。因此, 选择排序和插入排序的运行时间都为 $O(n^2)$ 。

尽管选择排序和插入排序相似，它们实际上还是有某些有趣的区别。例如，对于选择排序，要选出最小值，第二阶段中的每一步都需要扫描整个优先队列，这总是花费 $\Omega(n^2)$ 的时间。而插入排序的运行时间则取决于输入序列。例如，如果输入序列 S 是逆序，则插入排序的运行时间为 $O(n)$ 。

2.4.3 堆数据结构

对于插入和删除都是有效的优先队列的实现利用称为堆（heap）的数据结构。这种数据结构进行插入和删除操作的时间为对数时间。堆数据结构改进运行时间的基本方式是：它没有将元素和关键字存储在序列中，而是存储在一棵二叉树中。

堆（如图2-22所示）是一棵二叉树，它将关键字集合存储在其内部结点中，并且满足另外两个性质：一是关系性，按照关键字存储在 T 中的方式定义；二是结构性，按照 T 自身定义。假设，例如，通过比较器给定了关键字的全序关系。同样在堆的定义中， T 中的外部结点并不存储关键字或者元素，而仅作为“占位符”。 T 的关系性如下：

- **堆序性质**：在一个堆 T 中，对于除根之外的每个结点 v ，存储在 v 中的关键字大于或等于存储在 v 的父结点中的关键字。

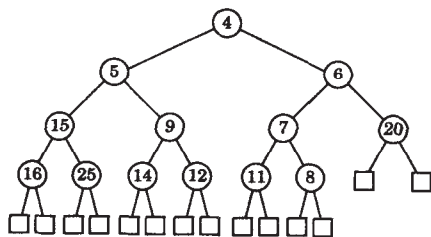


图2-22 存储13个整型关键字的堆示例。最后一个结点存储关键字8，外部结点为空

因此，从根至 T 的一个外部结点的路径上遇到的关键字按照非降序排列。同样，最小的关键字总是存储在根中。出于效率考虑，希望堆 T 的高度尽可能低。强制要求堆 T 满足另一结构性性质：

- **完全二叉树**：高度为 h 的一棵二叉树 T 是完全的（complete），如果层 $0, 1, 2, \dots, h-1$ 上的结点数达到最大（即层 i 有 2^i 个结点， $0 \leq i \leq h-1$ ）。且在第 $h-1$ 层，所有内部结点都在外部结点的左边。

通过指定 $h-1$ 层的所有内部结点都在外部结点的左边，意味着这一层的所有内部结点将按照中序遍历次序在该层的外部结点之前被访问。如图2-22所示。

通过强制要求堆 T 是完全的，可以识别堆 T 中除根之外的另一重要结点，即 T 的最后一个结点（last node），我们将这个结点定义成 T 中最右边、最深的内部结点。如图2-22所示。

1. 利用堆实现优先队列

基于堆的优先队列由以下几部分组成（如图2-23所示）：

- **heap**：是一棵完全二叉树，它的元素存储在内部结点中，其关键字满足堆序性质。假设利用2.3.4节描述的向量实现二叉树 T 。对于 T 中的每个内部结点 v ，用 $k(v)$ 表示存储在 v 中元素的关键字。
- **last**：对 T 中最后一个结点的引用。给定 T 的向量实现，假设实例变量 $last$ 是一个整型量，表示存储 T 中最后一个结点的向量中的单元索引。
- **comp**：是一个比较器，定义关键字之间的全序关系。不失一般性，假设 $comp$ 将最小元素保持在根结点中。如果希望将最大元素放在根结点中，那么可以相应地重新定义比较规则。

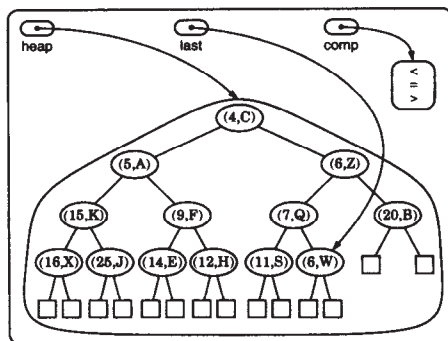


图2-23 基于堆的优先队列，存储整型关键字和文本元素

该实现的效率基于以下事实。

定理2.4 存储 n 个关键字的堆 T 的高度为 $h = \lceil \log(n+1) \rceil$ 。

证明 因为 T 是完全的， T 中内部结点个数至少为

$$1 + 2 + 4 + \dots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}$$

当第 $h-1$ 层只有一个内部结点时，达到这个下界。另一方面，观察 T 中内部结点个数至多为

$$1 + 2 + 4 + \dots + 2^{h-1} = 2^h - 1$$

当第 $h-1$ 层的所有 2^{h-1} 个结点都是内部结点时，达到这个上界。由于内部结点数等于关键字的个数 n ，因而， $2^{h-1} \leq n$ 且 $n \leq 2^h - 1$ 。因此，对这两个不等式的两边取对数，可得 $h \leq \log n + 1$ 和 $\log(n+1) \leq h$ ，这蕴涵着 $h = \lceil \log(n+1) \rceil$ 。 ■

因此，如果能够在堆上进行其运行时间与堆高度成正比的更新操作，那么这些操作将会有对数运行时间。

100

2. 用向量表示堆

当堆 T 可用向量实现时，最后一个结点的索引 w 总是等于 n 。第一个空的外部结点 z 的索引为 $n+1$ 。如图2-24所示。注意 z 的这个索引总是有效的，即使对于以下情况也是如此：

- 如果当前最后一个结点 w 是该层最右边的结点，那么 z 是最底层最左边的结点。如图2-24b所示。
- 如果 T 没有内部结点（即优先队列为空， T 中最后一个结点未定义），那么 z 是 T 的根。

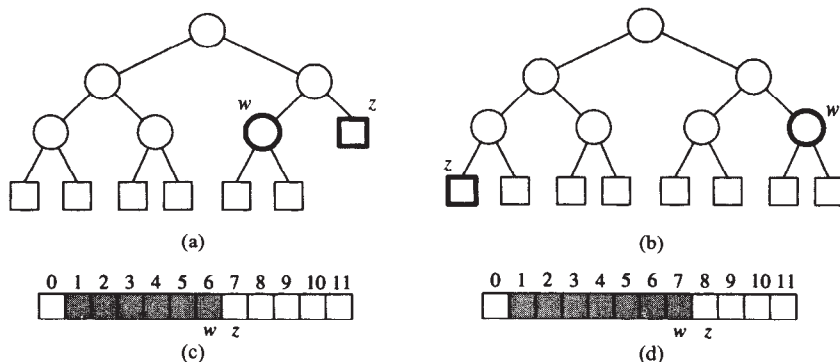


图2-24 堆中最后一个结点 w 和第一个外部结点 z ：(a)正常情况下， z 在 w 的右边；(b) z 在底层最左边的情况。(a)的向量表示如(c)所示；类似地，(b)的向量表示如(d)所示

用向量表示堆 T 所带来的简化可以帮助方法实现优先队列ADT。例如，更新方法`expandExternal(z)`和`removeAbove(z)`所需时间也为 $O(1)$ （假设不必扩展向量），因为它们只是涉及在向量中分配或释放单一单元。借助这个数据结构，方法`size`和`isEmpty`所需时间同样为 $O(1)$ 。此外，通过访问存储在堆的根（它在向量中的位序为1）中的元素或关键字，方法`minElement`和`minKey`也能轻松地在 $O(1)$ 时间内执行完成。而且，由于 T 是一棵完全二叉树，在一棵基于向量实现的二叉树中，关联堆 T 的向量有 $2n+1$ 个元素，按约定，其中的 $n+1$ 个是作为占位符的外部结点。实际上，因为所有外部结点的索引都比任何内部结点大，因此不必显式存储所有外部结点。如图2-24所示。

3. 插入

考虑如何利用堆 T ，执行优先队列ADT的方法`insertItem`。为了在 T 中存储一个新的关键字-元素对 (k, e) ，需要向 T 中添加一个新的内部结点。为了使 T 仍是一棵完全二叉树，必须添加新结点，使它成为 T 中新的最后一个结点。也就是说，在进行`expandExternal(z)`操作的地方，必须区分正确的外部结点 z ，该操作用一个内部结点（带有空外部子结点）替代 z ，然后在 z 处插入新元素。如图2-25a~b所示。称结点 z 为插入位置（insertion position）。

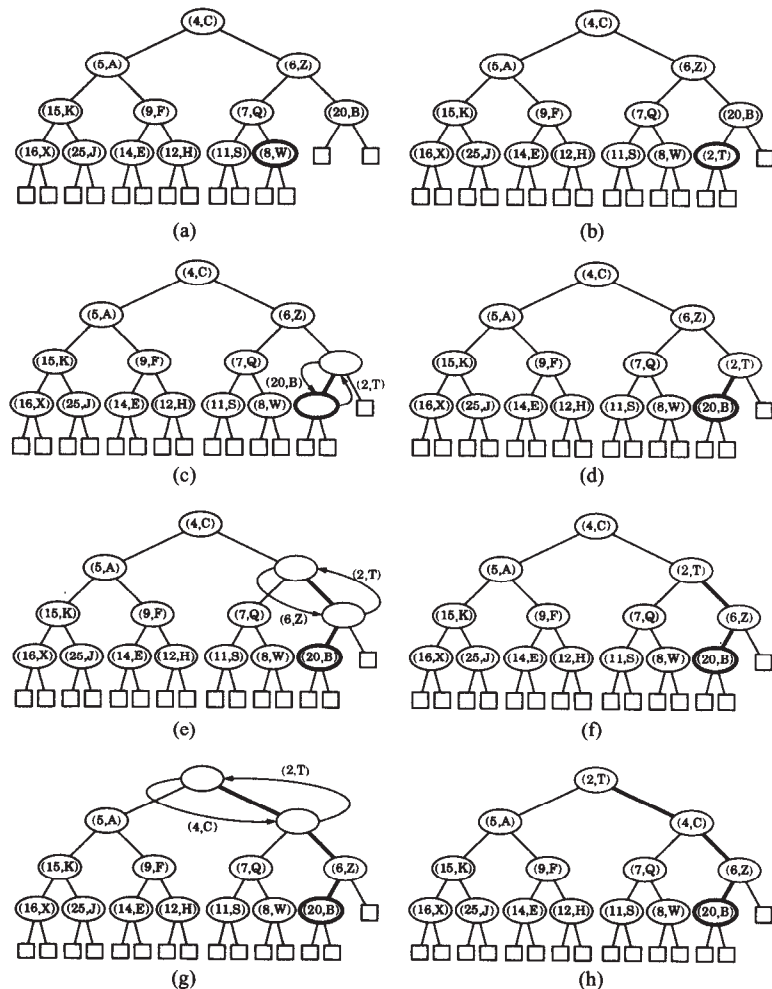


图2-25 将关键字为2的新元素插入图2-23的堆中：(a)初始堆；(b)将新的最后一个结点添加到原最后一个结点的右边；(c)~(d)交换以局部恢复堆序性质；(e)~(f)另一次交换；(g)~(h)最终交换

通常, 结点 z 是紧接在最后一个结点 w 右边的外部结点。如图2-24a所示。在任何情况下, 根据 T 的向量实现, 插入位置 z 被存储在索引 $n+1$ 处, 其中 n 为当前堆大小。因此, 对于向量实现的 T , 能够在常量时间内确定结点 z , 在执行 $\text{expandExternal}(z)$ 之后, 结点 z 成为最后一个结点, 我们将新的关键字-元素对 (k, e) 存储在 z 中, 于是 $k(z) = k$ 。

4. 插入后堆向上冒泡

在完成上述过程之后, 树 T 是完全的, 但它可能违反堆序性质。因此, 除非结点 z 是 T 的根 (即在插入之前, 优先队列为空), 否则将关键字 $k(z)$ 与存储在 z 的父结点 u 中的关键字 $k(u)$ 进行比较。如果 $k(u) > k(z)$, 那么需要恢复堆序性质, 可以通过交换存储在 u 和 z 处的关键字-元素对实现局部满足堆序性质。如图2-25c~d所示。这个交换导致新的关键字-元素对 (k, e) 上移一层。这可能再次违反堆序性质, 继续在 T 中向上交换, 直到没有违反堆序性质的情况出现。如图2-25e~h所示。

通常称通过交换向上移动的过程为堆向上冒泡 (up-heap bubbling)。一次交换可能解决违反堆序性质的问题, 或者向上传播一层。在最坏情况下, 堆向上冒泡导致新的关键字-元素对沿着通向堆 T 的根的路径移动。如图2-25所示。因此, 在最坏情况下, 方法 insertItem 的运行时间与 T 的高度成正比, 即为 $O(\log n)$, 因为 T 是完全的。

如果用向量实现 T , 那么可用 $O(1)$ 时间立即找到新的最后一个结点 z 。例如, 可以扩展一棵基于向量实现的二叉树, 以便添加一个方法返回索引为 $n+1$ 的结点, 即层编号为 $n+1$, 如2.3.4节中的定义。此外, 甚至可以定义一个 add 方法, 它将一个新元素添加到第一个外部结点 z 中, 该结点在向量中的位序为 $n+1$ 。在本章后面显示的代码段2-3中, 表明了如何利用这个方法高效地实现方法 insertItem 。另一方面, 如果用链表结构实现堆 T , 那么就要涉及更多关于查找插入位置 z 的过程 (参见习题C-2.27)。

5. 删除

现在讨论优先队列ADT的方法 removeMin 。利用堆 T 执行方法 removeMin 的算法如图2-26所示。

我们知道具有最小关键字的元素存储在堆 T (即使存在多个最小关键字) 的根 r 中, 然而, 除非 r 是 T 的唯一内部结点, 否则, 不能简单地删除结点 r , 因为这种操作将破坏二叉树的结构。作为替代, 访问 T 的最后一个结点 w , 并将它的关键字-元素对复制到根 r 中, 然后通过执行更新操作 $\text{removeAboveExternal}(u)$ 删除最后一个结点, 其中 $u = T.\text{rightChild}(w)$ 。这个操作会删除 u 的父结点 w 以及结点 u 自身, 并用它的左子结点代替结点 w 。如图2-26a~b所示。

进行这个常量时间的操作之后, 通过引用实现树 T 的向量中位序为 n (在删除之后) 的结点, 即可更新对最后一个结点的引用。

6. 删除后堆向下冒泡

但是, 现在任务还没有完成, 因为即使 T 现在是完全的, T 也可能违反堆序性质。为了确定是否需要恢复堆序性质, 检查 T 的根 r 。如果 r 的两个子结点是外部结点, 那么堆序性质得到满足, 并且完成了任务。否则, 区分两种情况:

- 如果 r 的左子结点是内部结点, 右子结点是外部结点, 设 s 是 r 的左子结点。
- 否则 (r 的两个子结点都是内部结点), 设 s 是 r 的子结点中具有最小关键字的结点。

如果存储在 r 处的关键字 $k(r)$ 大于存储在 s 处的关键字 $k(s)$, 那么需要恢复堆序性质。通过交换存储在 r 和 s 处的关键字-元素对可实现局部满足堆序性质。如图2-26c~d所示。注意不能交换 r 与 s 的兄弟结点。我们进行的交换恢复了结点 r 及其子结点的堆序性质, 但它可能违反 s 处的堆序性质; 因此, 可以继续沿着 T 向下交换, 直到不再出现违反堆序性质的情况, 如图2-26e~图2-26h所示。

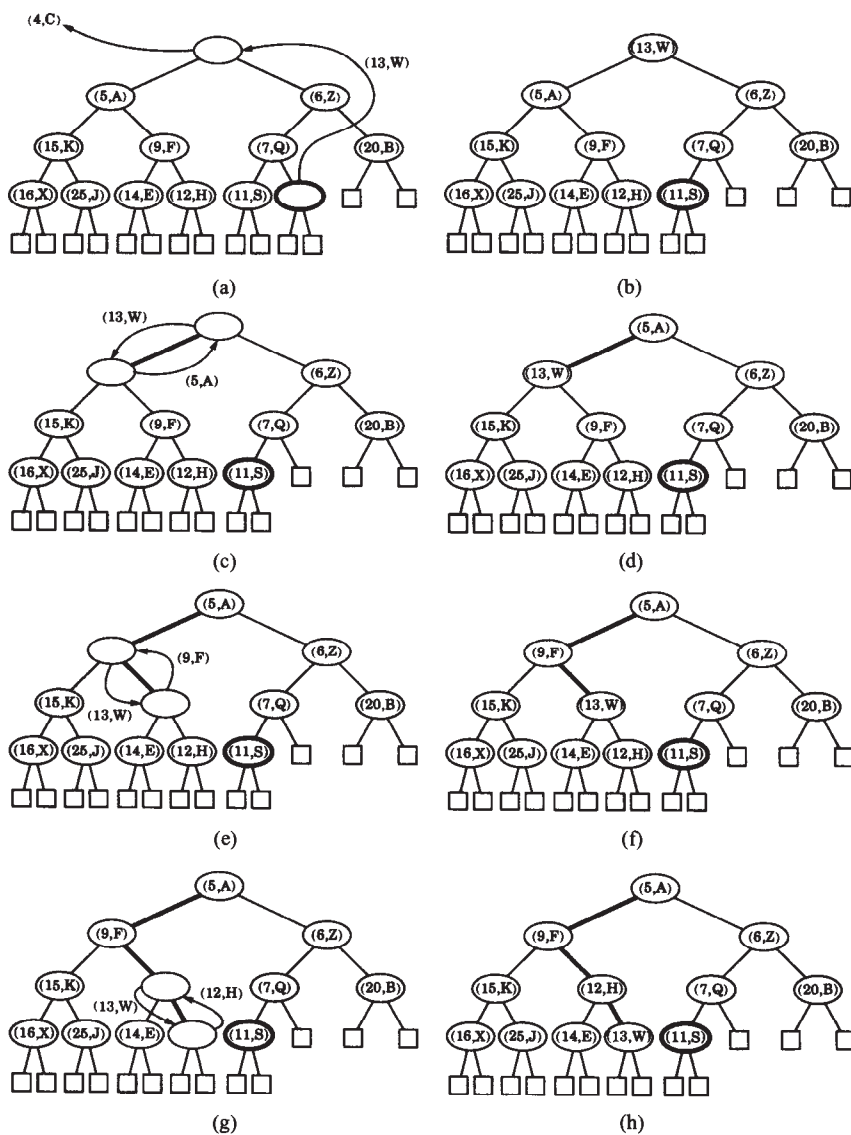


图2-26 从堆中删除具有最小关键字的元素：(a)~(b)删除最后一个结点，其关键字-元素对被存储在根中；(c)~(d)交换以局部恢复堆序性质；(e)~(f)另一次交换；(g)~(h)最终交换

称这个向下的交换过程为堆向下冒泡（down-heap bubbling）。一个交换要么解决了违反堆序性质的问题，或者向下传播一层。在最坏情况下，关键字-元素对将向下移动所有路径，直到底层的上一层。如图2-26所示。因此，在最坏情况下，方法removeMin的运行时间与堆T的高度成正比，即 $O(\log n)$ 。

7. 性能

表2-5给出了优先队列的堆实现的优先队列ADT方法的运行时间,假设堆 T 由二叉树的数据结构实现,这个数据结构在 $O(1)$ 时间支持二叉树ADT的方法(除了`element()`之外)。2.3.4节的链表结构和基于向量的结构容易满足这个要求。

表2-5 基于堆实现的优先队列的性能，而堆又由基于向量的二叉树结构实现。 n 表示方法执行时优先队列中的元素个数。如果堆是由链表结构实现的，则方法所需空间为 $O(n)$ ，如果堆是由基于向量的结构实现的，则方法所需空间为 $O(N)$ ，其中 $N(\geq n)$ 为实现向量所用的数组大小

操 作	时 间
size、isEmpty	$O(1)$
minElement、minKey	$O(1)$
insertItem	$O(\log n)$
removeMin	$O(\log n)$

简言之，优先队列ADT的每个方法都可在 $O(1)$ 或 $O(\log n)$ 时间内执行完成，其中 n 为方法执行时优先队列中的元素个数。对方法的运行时间分析基于如下几个方面：

- 因为 T 是完全的，因而堆 T 的高度为 $O(\log n)$ 。
- 在最坏情况下，堆向上冒泡和堆向下冒泡所需时间与堆 T 的高度成正比。
- 执行insertItem查找插入位置以及执行removeMin更新最后一个结点位置都需要常量时间。
- 堆 T 有 n 个内部结点，每个结点存储一个指向关键字的引用和一个指向元素的引用， T 还有 $n+1$ 个外部结点。

由此可得，堆数据结构非常高效地实现了优先队列ADT。与堆是用链表结构还是用序列实现无关。堆实现对于插入和删除操作快速有效，不像基于序列的优先队列实现。基于堆实现优先队列的一个重要的效率成果是，加快了优先队列排序的速度，即基于堆的排序算法要比基于序列的插入排序和选择排序算法快得多。

106

2.4.4 堆排序

再次考虑2.4.2节的PQ-Sort排序模式，它利用优先队列 P 对序列 S 排序。如果利用堆实现优先队列 P ，那么在第一阶段中， n 个insertItem操作中的每一个所需时间都为 $O(\log k)$ ，其中 k 为当时堆中的元素个数。同样，在第二阶段中， n 个removeMin操作中的每一个所需时间都为 $O(\log k)$ ，其中 k 为当时堆中的元素个数。由于总是有 $k \leq n$ ，最坏情况下每个这样的操作运行时间均为 $O(\log n)$ 。因此，每一阶段所需时间均为 $O(n \log n)$ 。因此当用堆实现优先队列时，整个优先队列的排序算法的运行时间为 $O(n \log n)$ 。称这个排序算法为堆排序（heap-sort），以下定理总结了它的性能。

定理2.5 堆排序算法对有 n 个可比较元素的序列进行排序所需时间为 $O(n \log n)$ 。

回忆表1-2，强调堆排序算法的运行时间 $O(n \log n)$ 要比选择排序和插入排序算法的运行时间 $O(n^2)$ 好得多。此外，可以对堆排序算法执行若干修改，以提高它的实际性能。

1. 原位实现堆排序

如果用数组实现要进行排序的序列 S ，则可以加快堆排序速度，并利用序列 S 自身的某一部分存储堆，将所需空间减少一个常数因子。因此可以避免使用外部堆数据结构。可对算法执行如下修改来达到这一点：

(1) 利用一个逆比较器，它对应一个堆，其中堆顶存储最大的元素。在算法执行过程中的任何时间，利用 S 的直到位序 $i-1$ 的左边部分存储堆中的元素，并用 S 的从位序 i 到 $n-1$ 的右边部分存储序列中的元素。因此， S 的前 i 个元素（位序为 $0, \dots, i-1$ ）提供了堆的向量表示（通过修改堆的层编号使其从0开始，不是从1开始），即位序为 k 的元素大于或等于其位序为 $2k+1$ 和 $2k+2$ 的“子结点”。

(2) 在算法的第一阶段中, 从空堆开始, 一次一步, 从左向右移动位于堆和序列之间的边界。

107 在第 i 步中 ($i = 1, \dots, n$), 通过在位序 $i-1$ 处添加元素扩展堆。

(3) 在算法的第二阶段中, 从空序列开始, 一次一步, 从右向左移动位于堆和序列之间的边界。在第 i 步中 ($i = 1, \dots, n$), 从堆中删除最大的元素, 并将它存储在位序 $n-i$ 处。

上述对堆排序所做的改变称为原位的 (in-place), 因为除序列自身所需的存储空间外, 只利用了额外的常量空间。我们没有把元素转移到序列外, 然后再放回, 而只是简单地重排它们。图2-27说明了原位堆排序的过程。一般而言, 如果一个排序算法除了被排序对象自身所需的存储空间外, 只利用了常量的存储空间, 则称这个排序算法是原位的。实际上原位排序算法的优势在于这样的算法能够充分利用运行它的计算机的主存空间。

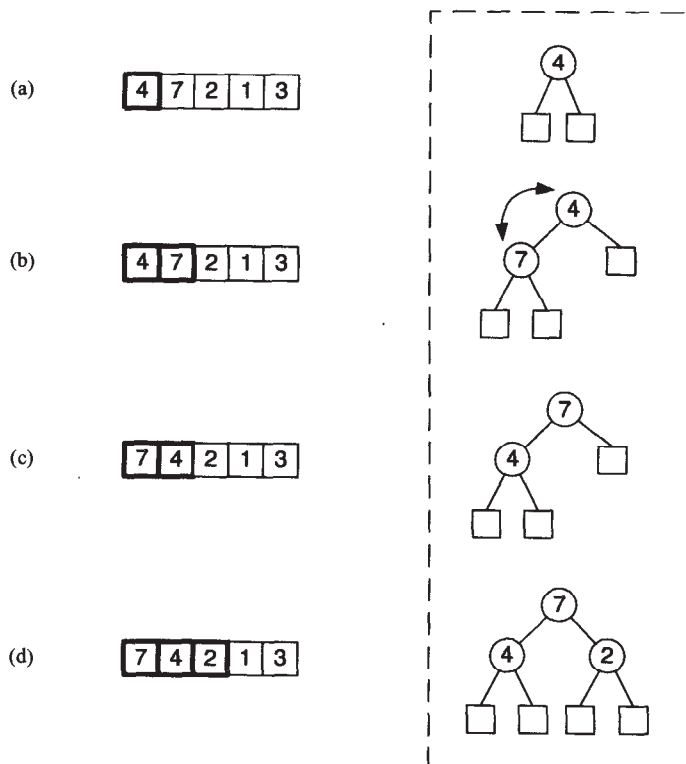


图2-27 原位堆排序第一阶段的前三步。粗线突出显示了向量中的堆部分。在向量旁边画出堆的二叉树视图, 即使这棵树实际上不是由原位算法所构造的

2. 自底向上构造堆

堆排序算法的分析表明, 通过 n 个连续的insertItem操作, 可以在 $O(n \log n)$ 时间内构造一个存储 n 个关键字-元素对的堆, 然后利用这个堆按序提取元素。然而, 如果预先给定所有要被存储在堆中的关键字, 则还有另一种运行时间为 $O(n)$ 的自底向上 (bottom-up) 的构造方法。

本节描述这个方法。观察可见, 它可被包含在Heap类中作为一个构造函数, 而不是利用 n 个insertItem操作序列填充堆。为简单起见, 假设关键字个数 n 是一个整数, 即

$$n = 2^h - 1$$

我们描述这个自底向上的构造方法, 即堆是一棵每层都被充满的完全二叉树, 因而堆高度为

$$h = \log(n+1)$$

用递归算法描述自底向上构造堆的过程。如算法2-16所示。调用该递归算法时，需要传递希望为其构建堆的存储关键字的序列。我们把该构造算法描述成作用在关键字上，并理解其元素伴随着关键字，即存储在树 T 中的项是关键字-元素对。

算法2-16 递归自底向上构造堆

算法 BottomUpHeap(S):

输入: 存储 $n = 2^h - 1$ 个关键字的序列 S

输出: 存储 S 中关键字的堆 T

if S 为空 **then**

return 空堆 (堆中只有一个外部结点)

从 S 中删除第一个关键字 k

将 S 分成两个序列 S_1 和 S_2 ，每个的大小为 $(n-1)/2$

$T_1 \leftarrow \text{BottomUpHeap}(S_1)$

$T_2 \leftarrow \text{BottomUpHeap}(S_2)$

建立存储 k 的根为 r 的二叉树 T ，左子树为 T_1 ，右子树为 T_2

如果需要，从 T 的根 r 处开始，进行堆向下冒泡

return T

称这个构造算法为“自底向上”堆构造，其原因是每次递归调用都会返回一棵子树，这棵子树是用于其存储元素的堆。也就是说，在其外部结点开始 T 的“堆化”，当每次递归调用返回时，沿着树向上前进。由于这个原因，有些作者将自底向上的堆构造称为“堆化”操作。

109

图2-28说明了 $h = 4$ 时自底向上构造堆的过程。

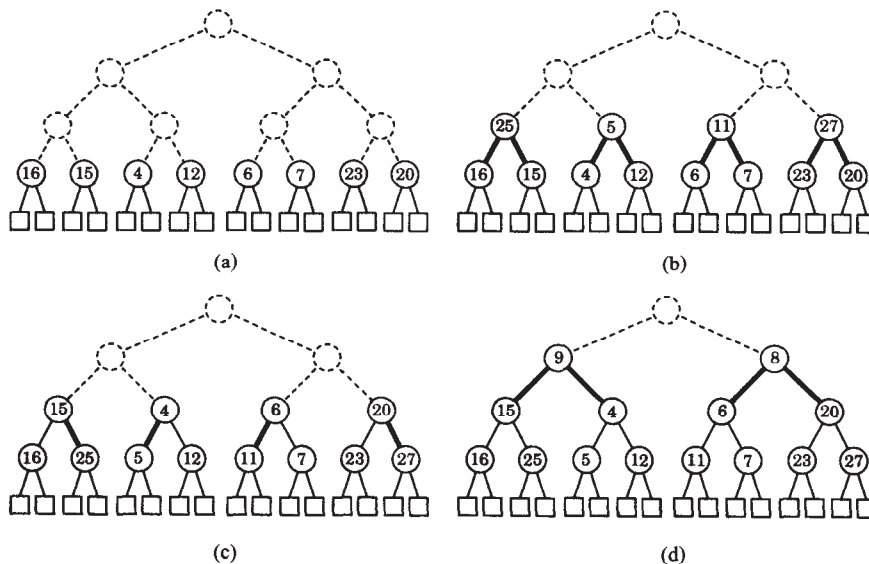


图2-28 自底向上构造有15个关键字的堆: (a)从底层开始构造有一个关键字的堆; (b)~(c)将这些堆组合成有3个关键字的堆; 然后(d)~(e)组合成有7个关键字的堆, 直到(f)~(g)建立最终的堆。粗线突出显示了堆向下冒泡的路径

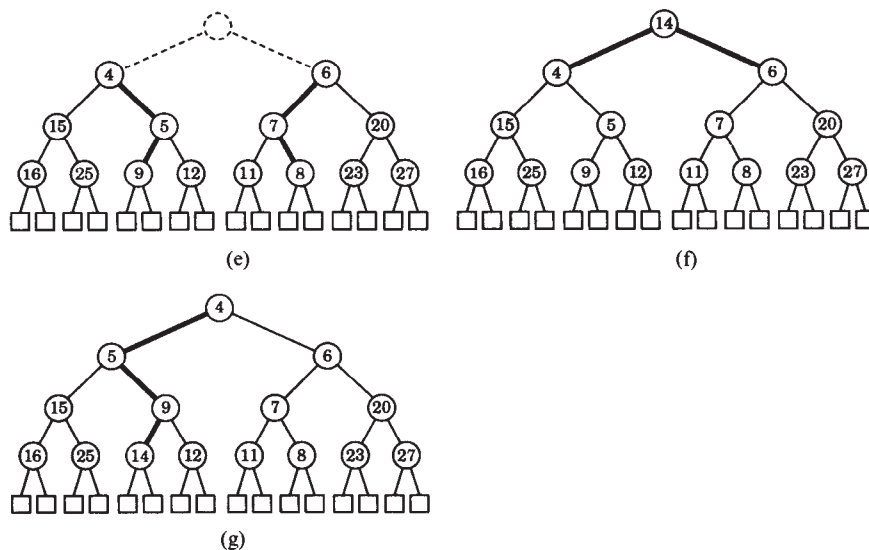


图2-28 (续)

110

自底向上堆构造比逐渐将 n 个关键字插入初始为空的堆中要快。正如以下定理表明的那样。

定理2.6 自底向上构造有 n 个项的堆所需时间为 $O(n)$ 。

证明 利用“可视化”方法分析自底向上堆构造的过程，如图2-29所示。

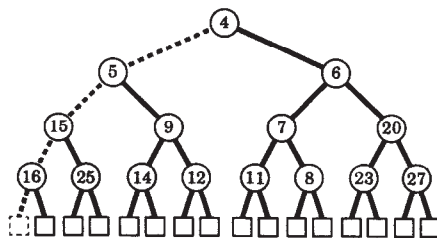


图2-29 自底向上堆构造的线性运行时间的可视化证明过程，其中与内部结点关联的路径已经交替用灰色和黑色突出显示。例如，与根结点关联的路径由存储关键字4、6、7和11的内部结点以及一个外部结点组成

设 T 是最终堆， v 是 T 的一个内部结点， $T(v)$ 表示根为 v 的 T 的子树。在最坏情况下，从两个递归形成的以 v 的子结点为根的子树来构造 $T(v)$ 的时间与 $T(v)$ 的高度成正比。当从 v 利用堆向下冒泡方式遍历一条路径，从 v 一直到 $T(v)$ 最底层的外部结点时，将会出现最坏情况。现在考虑 T 的从结点 v 到其中序后继外部结点的路径 $p(v)$ ，即这条路径开始在于 v ，转到 v 的右子结点，然后向左下方，直到到达一个外部结点。称路径 $p(v)$ 关联于（associate with）结点 v 。注意 $p(v)$ 不必是形成 $T(v)$ 时，堆向下冒泡方式得到的路径。显然， $p(v)$ 的长度（边数）等于 $T(v)$ 的高度。于是，在最坏情况下，形成 $T(v)$ 所需时间与 $p(v)$ 的长度成正比。因此，自底向上堆构造的总运行时间与关联于 T 的内部结点的路径长度之和成正比。

注意对于 T 中的任何两个内部结点 u 和 v ，路径 $p(u)$ 和路径 $p(v)$ 并不共享边，尽管它们可能共享结点。如图2-29所示。于是，关联于 T 的内部结点的路径长度之和不会超过堆的边数，即不超过 $2n$ 。由此可得，自底向上构造堆 T 所需时间为 $O(n)$ 。 ■

总之，定理2.6表明，堆排序的第一阶段可用 $O(n)$ 时间完成。不幸的是，最坏情况下堆排序第二阶段的运行时间为 $\Omega(n \log n)$ 。第4章将证明这个下界。

111

3. 定位器设计模式

本节讨论一种设计模式，它允许对优先队列ADT进行扩展，使其具有其他功能。例如，这些功能将会用于本书后面讨论的图算法中。

正如我们在分析表和二叉树时所看到的，将位置信息抽象到容器中是一种十分强大的工具。2.2.2节描述的位置ADT，能够识别容器中存储一个元素的特定“位置”。位置可以改变它的元素，例如，作为swapElements操作的结果，元素会发生变化，而位置保持不变。

还有一些应用，当在一个容器内部移动元素时，需要记录它们。满足这种需要的设计模式是定位器（locator）。定位器是一种机制，用于维持容器中一个元素和其当前位置之间的关联关系。定位器可以“粘住”一个特定元素，即使这个元素在容器中的位置发生改变也是如此。

定位器就像一个外套检查：我们可以把外套交给衣帽间的服务员，并收回的外套检查，它是我们外套的一个定位器。我们外套的位置相对其他外套的位置可以改变，因为其他外套可以被添加和删除，但总是可以利用我们的外套检查来检索我们的外套。要记住的关于定位器的重要事情是，它尾随在其数据项之后，即使它改变位置也是如此。

就像外套检查，现在想象当在容器中插入一个元素时可以取回某些信息——可以取回那个元素的定位器。这个定位器以后反过来又可用于引用容器内的元素。例如，指定应当从容器中删除的元素。作为一种抽象数据类型，定位器 ℓ 支持以下方法：

- `element()`：返回关联于 ℓ 的数据项的元素。
- `key()`：返回关联于 ℓ 的数据项的键。

为了表明正确性，以下讨论如何利用定位器扩展优先队列ADT的操作集，使其包括返回定位器以及将定位器作为参数的方法。

4. 基于定位器的优先队列方法

在优先队列的环境中利用定位器是很自然的方式。在这种场景中，定位器与插入优先队列中的数据项保持关联，并且允许用泛型方式访问数据项，而与优先队列的特定实现无关。对于优先队列的实现，这种能力是重要的，因为优先队列中没有`per`、`se`位置，因不用符号“位序”、“下标”或“结点”引用数据项。

112

5. 扩展优先队列ADT

利用定位器，可以扩展带有以下方法的优先队列ADT，这些方法用于访问和修改优先队列 P ：

- `min()`：返回 P 中具有最小关键字的数据项的定位器。
- `insert(k, e)`：将具有元素 e 和关键字 k 的新数据项插入 P 中，并返回该数据项的定位器。
- `remove(ℓ)`：从 P 中删除定位器为 ℓ 的数据项。
- `replaceElement(ℓ, e)`：用 e 替换定位器为 ℓ 的数据项中的元素，并返回原数据项中的元素。
- `replaceKey(ℓ, k)`：用 k 替换 P 中定位器为 ℓ 的数据项的关键字，并返回原数据项的关键字。

基于定位器的访问的运行时间为 $O(1)$ ，而基于关键字访问的运行时间在最坏情况下为 $O(n)$ ，因为这种访问必须在整个序列或堆中查找元素。此外，有些应用要求限制操作`replaceKey`，以使它只能增加或减少关键字。例如，可以定义新方法`increaseKey`或`decreaseKey`来实施这种限制，这两个方法把定位器作为参数。第7章将进一步讨论这种优先队列方法的应用。

6. 比较不同的优先队列实现

表2-6对基于无序序列、有序序列和堆实现的优先队列ADT方法（本节中定义了它们）的运行时间进行了比较。

表2-6 基于无序序列、有序序列和堆实现的优先队列ADT方法的运行时间比较。 n 表示方法执行时优先队列中的元素个数

方 法	无序序列	有序序列	堆
size、isEmpty、key、replaceElement	$O(1)$	$O(1)$	$O(1)$
minElement、min、minKey	$O(n)$	$O(1)$	$O(1)$
insertItem、insert	$O(1)$	$O(n)$	$O(\log n)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$

113

2.5 字典与散列表

计算机字典类似于纸质字典，在某种意义上，它们都用于进行查找。其主要思想是用户可以将关键字赋予元素，然后利用这些关键字查找或删除元素。如图2-30所示。因此，字典抽象数据类型有用于插入、删除和查找具有某个关键字的元素的方法。

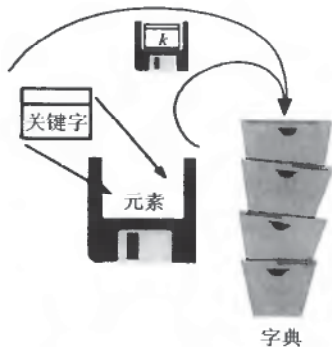


图2-30 字典ADT的概念说明。关键字（标签）被用户赋予元素（磁盘）。结果数据项（标记的磁盘）被插入到字典中（文件夹）。关键字以后可用于检索或者删除数据项

2.5.1 无序字典 ADT

字典存储关键字-元素对 (k, e) 。称之为数据项（item），其中 k 为关键字， e 为元素。例如，在一个存储学生记录（如学生名、地址和课程分数）的字典中，关键字可能是学生的ID号。在某些应用中，关键字可能是元素自身。

字典有两种类型：无序字典（unsorted dictionary）和有序字典（sorted dictionary）。第3章研究有序字典；这里讨论无序字典。无论哪一种情况，都利用关键字（key）作为标识符，它被应用或者用户赋予关联的元素。

为了具有通用性，定义的字典可以存储多个具有相同关键字的数据项。然而，有些应用不希望数据项具有相同的关键字（例如，在一个存储学生记录的字典中，不允许两个学生具有相同的ID）。在这样的情况下，当关键字唯一时，关联某个对象的关键字可看作那个对象在内存中的“地址”。有时称这样的字典为“关联存储”，因为关联于对象的关键字确定了它在字典中的“位置”。

114

作为一种ADT，字典（dictionary） D 支持以下基本方法：

- **findElement(k)**: 如果 D 包含关键字等于 k 的数据项, 那么返回该数据项的元素, 否则返回一个特殊元素NO_SUCH_KEY。
- **insertItem(k, e)**: 将元素为 e 、关键字为 k 的数据项插入 D 中。
- **removeElement(k)**: 从 D 中删除关键字等于 k 的数据项, 并返回其元素。如果 D 中没有该数据项, 那么返回一个特殊元素NO_SUCH_KEY。

注意: 如果想要将数据项 e 存储在字典中, 使得该数据项是其自身的关键字, 那么可以调用方法insertItem(e, e)插入 e 。当操作findElement(k)和removeElement(k)不成功时 (即字典 D 中不含关键字等于 k 的数据项), 根据约定返回一个特殊元素NO_SUCH_KEY。称这样一个特殊元素为观察哨 (sentinel)。

此外, 字典可以实现容器的其他支持的方法, 如size()和isEmpty()方法。此外, 还可以包含方法elements(), 它返回存储在 D 中的元素; 以及方法key(), 它返回存储在 D 中的关键字。同样, 允许关键字不唯一可以激发包含像findAllElements(k)和removeAllElements(k)这样的方法, 前者用于返回关键字等于 k 的所有元素的迭代器, 后者用于从 D 中删除关键字等于 k 的所有数据项, 并返回这些元素的一个迭代器。

日志文件

实现字典 D 的一种简单实现方式是利用无序序列 S , 而 S 是用向量或表实现的, 用来存储关键字-元素对。这样的一种实现常常称为日志文件 (log file) 或审计线索 (audit trail)。日志文件主要应用在存储少量数据元素或者数据不太可能随时间大量改变的地方。我们也称 D 的日志文件实现为无序序列实现 (unordered sequence implementation)。

日志文件所需空间为 $\Theta(n)$, 因为向量和链表数据结构能够维持的存储空间与其大小成正比。此外, 用日志文件实现字典ADT, 能够容易、高效地实现操作insertItem(k, e), 正像调用 S 上的方法insertLast一样, 所用时间为 $O(1)$ 。

不幸的是, 执行findElement(k)操作要扫描整个序列 S , 从而检查每个数据项。当查找不成功时, 到达序列末尾, 这会检查所有 n 个数据项, 此时运行这个方法显然就会出现最坏情况。因此, findElement方法的运行时间为 $O(n)$, 类似地, 在 D 上运行removeElement(k)方法最坏情况下需要线性时间, 这是因为要删除给定关键字的一个数据项, 必须首先扫描整个序列 S 以找到它。

115

2.5.2 散列表

字典中关联元素的关键字常常表示这些元素的“地址”。这类应用的例子有编译器的符号表和环境变量的注册表。这两种结构都由符号名集合组成, 其中每个名字充当关于变量类型和值的属性的“地址”。为了实现这种环境中的字典, 使用散列表 (hash table) 是一种最高效的方式。正如将要看到的那样, 使用散列表时, 尽管字典ADT操作最坏情况下的运行时间为 $O(n)$, 其中 n 为字典中的数据项数, 但它的期望时间为 $O(1)$ 。散列表主要由两部分组成, 第一部分是桶数组。

1. 桶数组

散列表的桶数组 (bucket array) 是一个大小为 N 的数组 A , 其中 A 的每个单元可看作一只“桶” (即关键字-元素对的容器), 整数 N 表示数组的容量 (capacity)。如果关键字为整数, 且均匀分布在范围 $[0, N-1]$ 中, 这个桶数组就是所需的数组。关键字为 k 的元素 e 被简单地插入到桶 $A[k]$ 中。如果关联某些关键字的桶单元没有出现在字典中, 则假设该单元保存了特殊的NO_SUCH_KEY对象, 如图2-31所示。

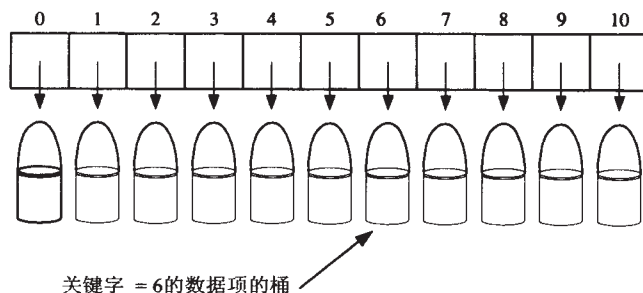


图2-31 桶数组说明

当然，如果关键字不是唯一的，那么两个不同的元素可能被映射到 A 中的同一只桶。在这种情况下，就称发生了冲突（collision）。显然，如果 A 的每只桶只能存储一个元素，那么我们在一只桶内就不能关联多个的元素。在发生冲突的情况下，这是一个问题。但要确信，有许多处理冲突的方法，将在稍后讨论它们，但最好的策略是在第一个地方设法避免冲突。

2. 桶数组结构分析

如果关键字是唯一的，那么冲突就不需要关注，并且在散列表中进行查找、插入和删除操作的时间最坏情况下为 $O(1)$ 。这听起来就像一个伟大的成就，但它有两个缺点。第一个缺点是它利用 $\Theta(N)$ 的空间，这未必与实际出现在字典中的数据项数 n 有关。如果 N 相对于 n 比较大，那么这种实现就会浪费空间。第二个缺点是桶数组要求关键字是位于 $[0, N-1]$ 区间内的唯一整数。实际上经常不是这样。因为这两个缺点是如此普遍，所以把散列表的数据结构定义成由桶数组以及从关键字到区间 $[0, N-1]$ 内的整数的“良好”映射组成。

2.5.3 散列函数

散列表结构的第二部分是一个函数 h ，称为散列函数（hash function）。它将字典中的每个关键字 k 映射到区间 $[0, N-1]$ 内的一个整数，其中 N 是这个散列表的桶数组的容量。通过这样的散列函数 h ，能够将桶数组方法应用于任意关键字。这个方法的主要思想是利用散列函数值 $h(k)$ 作为桶数组 A 的下标，而不是使用关键字 k （对于作为桶数组下标，它最有可能是不合适的），即将数据项 (k, e) 存储在桶 $A[h(k)]$ 中。

如果一个散列函数在映射字典中的关键字时，能够尽可能地减少冲突，则称该函数是“良好的”。由于实际原因，我们也希望对给定散列函数的评估是计算快速、容易。按照惯例，将散列函数 $h(k)$ 的评估看作由两个行为组成——将关键字映射到一个整数上，称为散列编码（hash code）；将散列编码映射到桶数组下标内的一个整数上，称为压缩映射（compression map），如图2-32所示。

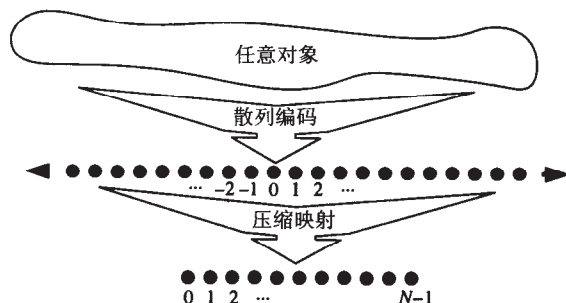


图2-32 散列函数的两个部分：散列编码和压缩映射

1. 散列编码

散列函数执行的第一个动作是取任意一个关键字 k ，并赋予它一个整型值。把赋予关键字 k 的整数称为 k 的散列编码 (hash code) 或散列值 (hash value)。这个整型值不一定在区间 $[0, N-1]$ 内，甚至可能为负数，但希望赋予关键字的散列编码集应该尽可能地避免冲突，此外，为了与所有关键字保持一致，用于关键字 k 的散列编码应该与等于 k 的任何关键字的散列编码相同。

2. 部分和

对于其位表示为散列编码位表示的两倍的基本类型，不能直接应用上述模式。尽管如此，一种可能的散列编码的确已被用于许多Java实现中，它简单地将类型的一个（长）整型表示向下强制转换成为大小为散列编码的一个整数。当然，这个散列编码忽略了出现在原始值中的一半信息，并且如果字典中许多关键字仅在这些位上不同，那么利用这种简单的散列编码就会出现冲突。另一种散列编码考虑所有初始位，将高阶位的整数表示与低阶位的整数表示相加，并把结果作为散列编码。可以将部分和方法扩展到任意对象 x 上，可将它的二进制表示看作一个 k 元整数组 $(x_0, x_1, \dots, x_{k-1})$ ，然后就可以形成 x 的一个散列编码 $\sum_{i=0}^{k-1} x_i$ 。

3. 多项式散列编码

上面描述的求和散列编码对于表征字符串或者其他形如 $(x_0, x_1, \dots, x_{k-1})$ 的多种长度的对象（可将其视作元组）不是一种好的选择，其中 x_i 的次序非常重要。例如，考虑字符串 s ，对 s 中字符的ASCII（或Unicode）值进行求和并将得到的结果作为它的散列编码。不幸的是，这个散列编码对于一批字符串产生了大量不希望的冲突。尤其是，利用这个函数，“temp01”和“temp10”会发生冲突，正像“stop”、“tops”、“pots”和“spot”也会发生冲突一样。更好的散列编码应该以某种方式考虑 x_i 的位置。另一种散列编码确实考虑了这一点。它选择一个非零常数 $a \neq 1$ ，利用以下多项式的值作为散列编码。

$$x_0 a^{k-1} + x_1 a^{k-2} + \dots + x_{k-2} a + x_{k-1}$$

利用Horner规则（见习题C-1.16），可将上式重写为

$$x_{k-1} + a(x_{k-2} + a(x_{k-3} + \dots + a(x_2 + a(x_1 + ax_0)) \dots))$$

从数学上讲，它是一个关于 a 的简单多项式，以对象 x 的分量 $(x_0, x_1, \dots, x_{k-1})$ 作为它的系数。因此，称这个散列编码为多项式散列编码 (polynomial hash code)。

118

4. 实验性的散列编码分析

直观上讲，多项式散列编码利用与常数 a 相乘为值的元组中的每个分量“产生空间”，同时还会保留以前的分量特征。当然，在一台典型的计算机上，利用散列编码的有限位表示，可以计算多项式的值。因此，该值将会周期性的溢出用于表示整数的位。因为我们对对象 x 关于其他关键字的良好展开更感兴趣，就简单地忽略了这样的溢出。我们仍然应该记住，这样的溢出正在发生，并且选择常数 a ，使其具有一些非零的低阶位，在溢出时它可以用于保存一些信息内容，甚至当我们处于溢出情形中时亦可如此。

我们已经进行了一些实验，研究表明对于由英语单词组成的字符串， a 为33、37、39和41是非常好的选择。事实上，在50 000个以上的英语单词表中，我们发现取 a 为33、37、39或41时，每种情况会产生少于7次的冲突。该单词表是由Unix的两个变体提供的单词表合并而成的。应该不会令人吃惊的是，许多实际字符串的实现利用 a 的其中一个常数，选择多项式散列函数作为字符串的默认散列编码。出于速度方面的考虑，有些实现仅将多项式散列函数应用于一个长字符串的一部分字符，比如说，每8个字符。

2.5.4 压缩映射

关键字 k 的散列编码通常不适合直接应用于桶数组，因为关键字的散列编码的范围通常会超过桶数组 A 的合法下标。也就是说，由于下标为负数或是超过数组 A 的容量，因此不正确地利用散列编码作为桶数组的下标可能导致抛出数组越界异常。因此，一旦确定关键字对象 k 的一个整型散列编码，还有一个将这个整数映射到区间 $[0, N-1]$ 上的问题。这个压缩步骤是散列函数执行的第二个步骤。

1. 除法散列法

将要使用的一种简单的压缩映射（compression map）是

$$h(k) = |k| \bmod N$$

119 它称为除法散列法（division method）。

如果取 N 为素数，那么除法压缩映射有助于“扩散”散列值的分布。实际上，如果 N 不是素数，关键字分布中的模式在散列编码分布中出现重复的可能性更大，因此会导致冲突。例如，如果将关键字 $\{200, 205, 210, 215, 220, \dots, 600\}$ 散列到大小为100的桶数组中，那么每个散列编码将与另外三个发生冲突。但是如果将这组相同的关键字散列到大小为101的桶数组中，则不会发生冲突。如果散列函数选择得好，将能保证两个不同的关键字散列到同一只桶中的概率至多为 $1/N$ 。但是，选择 N 为素数并不总是足够，因为对于若干个不同的 i ，如果关键字值中存在形如 $iN+j$ 的重复模式，则仍然会发生冲突。

2. MAD方法

更复杂的压缩函数可以消除整型关键字集中的重复模式，称之为乘加与除（multiply add and divide）或“MAD”方法。这种方法定义压缩函数如下

$$h(k) = |ak + b| \bmod N$$

其中 N 是素数， a 和 b 是在确定压缩函数时随机选取的非负整数，满足 $a \bmod N \neq 0$ 。选择这个压缩函数可以消除散列编码集中的重复模式，从而更接近于得到一个“良好”的散列函数，即任何两个不同的关键字发生冲突的概率至多为 $1/N$ 。这种良好的行为就如同将这些关键字随机均匀地“抛撒”到数组 A 中一样。

有了这样一个压缩函数，它将 n 个整数公平均匀的分布到区间 $[0, N-1]$ 中，并将字典中的关键字映射到整数，这样就有了一个有效的散列函数。这样的散列函数和桶数组一起定义了字典ADT的散列表实现的关键字部分。

但是，在给出执行诸如findElement、insertItem和removeElement之类的操作的细节之前，必须首先解决如何处理冲突的问题。

2.5.5 冲突处理模式

散列表的主要思想是取一个桶数组 A 和一个散列函数 h ，然后通过将每个数据项 (k, e) 存储在桶 $A[h(k)]$ 中，利用它们实现一个字典。然而，当两个不同的关键字 k_1 和 k_2 满足 $h(k_1) = h(k_2)$ 时，这个简单的思想受到了挑战。这种冲突的存在阻止了将一个新数据项 (k, e) 直接插入到桶 $A[h(k)]$ 中。

120 同时，它们还使执行findElement(k)操作的过程变得复杂。因此，需要用于解决冲突的一致策略。

1. 分离链地址法

处理冲突的一种简单、有效方法是，使每个桶 $A[i]$ 存储一个指向链表、向量或序列 S_i 的引用， S_i 中存储了所有数据项，散列函数已经将它们映射到桶 $A[i]$ 上。可将序列 S_i 看作一个由无序序列或日志文件（log file）方法实现的小型字典，但限制只存储那些满足 $h(k) = i$ 数据项 (k, e) 。称这种冲

冲突解决 (collision resolution) 规则为分离链地址法 (separate chaining)。假设用这种方式在小型字典中将每只非空桶实现为日志文件, 那么就能够执行以下基本的字典操作:

- findElement(k):
 - $B \leftarrow A[h(k)]$
 - if B 为空 then
 - return NO_SUCH_KEY
 - else
 - {在这只桶的序列中查找关键字 k }
 - return B .findElement(k)
- insertItem(k, e):
 - if $A[h(k)]$ 为空 then
 - 建立一个初始为空的、基于序列的新字典 B
 - $A[h(k)] \leftarrow B$
 - else
 - $B \leftarrow A[h(k)]$
 - B .insertItem(k, e)
- removeElement(k):
 - $B \leftarrow A[h(k)]$
 - if B 为空 then
 - return NO_SUCH_KEY
 - else
 - return B .removeElement(k)

因此, 对于涉及关键字 k 的每个基本字典操作, 将其委托给存储在 $A[h(k)]$ 中基于序列的小型字典处理。因此, 插入操作将把新的数据项放在序列的末尾; 查找操作将查遍整个序列, 直至到达序列末尾, 或者找到具有给定关键字的数据项; 删除操作会另外将找到的数据项删除。利用这种简单的日志文件字典实现, 在这些情况下可以侥幸获得成功, 因为散列函数的分散性有助于保持每个小型字典的大小足够小。的确, 一个好的散列函数将设法使冲突尽可能地少, 这蕴涵着大多数桶或者为空, 或者只存储了一个数据项。

121

在图2-33中, 说明了一个简单的散列表。它利用除法压缩函数和分离链地址法解决冲突。

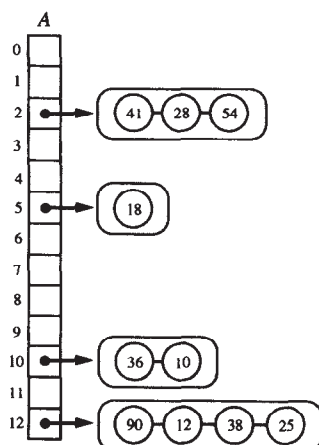


图2-33 大小为13的散列表示例, 其中存储有10个整型关键字, 用链地址法解决冲突。此示例中的压缩映射为 $h(k) = k \bmod 13$

2. 装填因子和再散列法

假设利用一个好的散列函数将字典中的 n 个数据项存储在容量为 N 的桶数组中，并期望每个桶的大小为 n/N 。称为散列表装填因子（load factor）的参数是一个常数，通常小于1。给定一个好的散列函数，对于散列表实现的字典，利用该函数执行操作findElement、insertItem和removeElement，期望运行时间为 $O(\lceil n/N \rceil)$ 。因此，如果假定 n 为 $O(N)$ ，实现标准字典操作的期望运行时间为 $O(1)$ 。

一般来说散列表的装填因子为0.75。当添加一个元素，导致超过这个界限时，保持散列表的装填因子为常数还需做其他工作。在这种情况下，为了保持装填因子小于某个指定的常数，需要增加桶数组的大小，并改变压缩映射使之适合于这个新的大小。此外，还必须利用这个新的压缩映射，将所有现有的散列表元素插入新的桶数组中。这样一个增加大小和重建散列表的过程称为再散列（rehashing）。按照1.5.2节可扩展数组的方法，一种良好的再散列法是将原数组大约增加一倍，并选择新数组的大小为素数。

122

3. 开放定址法

分离链地址法有许多美妙的性质，如可以简单地实现字典操作，但它有一个微小的缺点：需要使用辅助数据结构——表、向量或序列——将那些具有冲突关键字的数据项保存为日志文件。不过，除了利用分离链地址法之外，还可以利用其他方法处理冲突。如果空间非常宝贵，则可以采用另一种方法，总是将每个数据项直接存储在桶中，并且每只桶中至多存储一项。这种方法由于没有利用辅助数据结构而节省空间，但它需要更复杂的方法来处理冲突。有几种实现这种方法的方式，称为开放定址法（open addressing）。

4. 线性探测法

一种简单的开放定址处理冲突的策略是线性探测法（linear probing）。在这种策略中，如果试图向已被占据的桶 $A[i]$ 中插入一个数据项 (k, e) ，其中 $i = h(k)$ ，那么下一次将尝试在 $A[(i+1) \bmod N]$ 中插入数据项。如果 $A[(i+1) \bmod N]$ 中已被占据，那么将尝试 $A[(i+2) \bmod N]$ ，依此类推，直到在 A 中找到空桶，它可以接受该新的数据项。一旦定位桶的位置，则简单地将数据项 (k, e) 插入到该位置。利用这种解决冲突的策略，需要改变操作findElement(k)的实现。尤其是，进行这样的查找，需要从 $A[h(k)]$ 开始连续检查每个桶，直至找到关键字等于 k 的数据项，或者找到空桶（在这种情况下，查找不成功）。如图2-34所示。

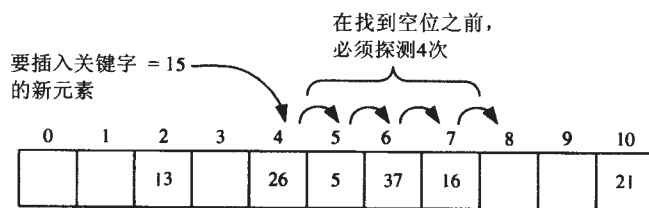


图2-34 利用线性探测法插入数据项到散列表中，以解决冲突。这里利用压缩映射 $h(k) = k \bmod 11$

然而，moveElement(k)比这更复杂。实际上，为了完全实现这个方法，应该恢复桶数组的内容，就像关键字为 k 的数据项从未在第一次插入到桶 $A[i]$ 中一样。尽管进行这样的恢复当然是可能的，但需要将桶中 $A[i]$ 以上的数据项向下移动，同时不移动这个组中的其他数据项（即已位于正确位置的数据项不需移动）。解决这个难点的典型方法是使用一个特殊的“惰性数据项”对象替换被删除的数据项。这个对象必须用某种方法标记，使得当它占据一个给定的桶时，可以快速检测到这一事实。利用这个可能占据散列表中的桶的特殊标记，修改查找算法removeElement(k)或

123

findElement(k), 使得对关键字 k 的查找能够跳过惰性数据项, 继续探测直至找到所需的数据项或空桶。但对于insertItem(k, e)算法, 相反应该在一个惰性数据项处停下来, 并用要插入的新数据项代替这个惰性数据项。

线性探测法节省空间, 但删除操作复杂。即使利用了惰性数据项对象, 线性探测解决冲突的策略仍然有其他缺点。它趋向于将字典中的数据项聚集成连续的区域, 导致查找过程变得相当慢。

5. 二次探测法

另一种开放定址策略称为二次探测法 (quadratic probing)。对于 $j = 0, 1, 2, \dots$, 它需要迭代地尝试桶 $A[(i + f(j)) \bmod N]$, 其中 $f(j) = j^2$, 直至找到一个空桶。正如线性探测法一样, 二次探测策略也使删除操作复杂, 但它避免了线性探测中出现的聚集模式。然而, 这种方法产生了自己的聚集, 称为二次聚集 (secondary clustering), 其中填充的数组单元集以固定模式沿着数组来回“弹跳”。如果 N 没有被选为素数, 那么即使 A 中存在空桶, 二次探测策略可能也找不到它。事实上, 即使选 N 为素数, 如果桶数组至少半满, 则二次探测策略可能也找不到空桶。

6. 双散列法

另一种开放定址策略不会引起由线性探测或二次探测导致的聚集, 这种方法是双散列 (double hashing) 策略。在这种方法中, 选择一个二级散列函数 h' , 并且如果 h 将关键字 k 映射到已被占据的桶 $A[i]$, 其中 $i = h(k)$, 那么对于 $j = 1, 2, 3, \dots$, 反复尝试桶 $A[(i + f(j)) \bmod N]$, 其中 $f(j) = j \cdot h'(k)$ 。在这种模式中, 二级散列函数的计算值不能为0; 一般选择 $h'(k) = q - (k \bmod q)$, $q (< N)$ 为某个素数。同时 N 也为素数。此外, 应该选择二级散列函数, 尽可能使聚集最小。

这些开放定址 (open addressing) 模式比分离链地址法节省空间。但是它们未必是更快的方法。在实验和理论分析中, 与其他方法相比, 链地址法具有竞争力, 或者运行更快, 这取决于桶数组的装填因子。因此, 如果内存空间不成问题, 处理冲突的方法似乎是链地址法。尽管如此, 如果内存空间不足, 假定探测策略能够使得开放定址法导致的聚集达到最小, 那么这些开放定址法是值得实现的。

124

2.5.6 通用散列

在这一节里, 将证明如何保证一个散列函数是良好的。为了仔细研究这个问题, 需要进行一点更数学化的讨论。

正如早先所提到的, 不失一般性地假设关键字集是某个范围的整数。设 $[0, M-1]$ 就是这个范围。因此, 可将一个散列函数看作是从区间 $[0, M-1]$ 内的整数到区间 $[0, N-1]$ 内的整数的映射。还可以将正考虑的候选散列函数集看作散列函数族 (family) H 。如果对于区间 $[0, M-1]$ 内的任何两个整数 j 和 k 和随机均匀从 H 中选择一个散列函数, 满足

$$\Pr(h(j) = h(k)) \leq 1/N$$

则称族 H 是通用的 (universal)。这样的族也称为2-通用 (2-universal) 散列函数族。因此, 可以把选择良好的散列函数的目标看作是选择一个易于计算的小通用散列函数族的问题。通用散列函数族有用的原因在于, 它们可导致较低的期望冲突数量。

定理2.7 设 j 是区间 $[0, M-1]$ 内的一个整数, S 是此区间内的 n 个整数集, h 是从通用散列函数族中随机均匀地选择的散列函数, 这些散列函数是从区间 $[0, M-1]$ 内的整数到区间 $[0, N-1]$ 内的整数的映射。那么, 整数 j 和 S 中的整数之间的期望冲突数量至多为 n/N 。

证明 令 $c_h(j, S)$ 表示整数 j 和 S 中的整数之间发生冲突的次数 (即 $c_h(j, S) = |\{k \in S: h(j) = h(k)\}|$)。我们感兴趣的是 $E(c_h(j, S))$ 的期望值。将 $c_h(j, S)$ 写成

$$c_h(j, S) = \sum_{k \in S} X_{j,k}$$

其中 $X_{j,k}$ 是随机变量, 如果 $h(j) = h(k)$, 则为 1, 否则为 0 (即 $X_{j,k}$ 是 j 和 k 之间发生冲突的指示器 (indicator) 随机变量)。由期望的线性关系可知,

$$E(c_h(j, S)) = \sum_{s \in S} E(X_{j,k})$$

同时, 由通用散列函数族的定义可知, $E(X_{j,k}) \leq 1/N$ 。因此,

$$E(c_h(j, S)) \leq \sum_{s \in S} \frac{1}{N} = \frac{n}{N} \quad \blacksquare$$

125

换一个角度讲, 这个定理指出散列编码 j 和已在散列表 (利用从通用散列函数族 H 中随机选择的一个散列函数) 中的关键字之间发生冲突的期望次数至多为散列表的当前装填因子。利用链地址法解决冲突, 对于关键字 j , 在散列表中进行查找、插入或删除所需时间与 j 和表中其他关键字之间发生冲突的次数成正比。这蕴涵着任何一个这样的操作的期望运行时间与散列表的装填因子成正比。这正是我们所希望的。

让我们把注意力转到如何构造一个小通用散列函数族的问题, 其中的散列函数是易于计算的。构造的散列函数集实际上类似于上一节最后考虑的目标族。设 p 是一个素数, 它大于或等于散列编码数 M , 小于 $2M$ (按照称为 Bertrand 假定的数学事实, 总是存在这样的素数)。

定义 H 为如下形式的散列函数集

$$h_{a,b}(k) = (ak + b \bmod p) \bmod N$$

下列定理表明这种散列函数族是通用的。

定理 2.8 散列函数族 $H = \{h_{a,b}: 0 < a < p \text{ 且 } 0 \leq b < p\}$ 是通用的。

证明 设 Z 表示区间 $[0, p-1]$ 上的整数集, 将每个散列函数 $h_{a,b}(k)$ 分成函数

$$f_{a,b}(k) = ak + b \bmod p$$

和

$$g(k) = k \bmod N$$

使得 $h_{a,b}(k) = g(f_{a,b}(k))$ 。函数集 $f_{a,b}$ 定义了散列函数族 F , 它将 Z 中的整数映射到 Z 中的整数。声明 F 中的每个散列函数根本不会引发冲突。为了证明这一点, 考虑 $f_{a,b}(j)$ 和 $f_{a,b}(k)$, j 和 k 是 Z 中的某一对不同的整数。如果 $f_{a,b}(j) = f_{a,b}(k)$, 则会有冲突。但是回忆模操作的定义, 这蕴涵着

$$aj + b - \left\lfloor \frac{aj+b}{p} \right\rfloor p = ak + b - \left\lfloor \frac{ak+b}{p} \right\rfloor p$$

不失一般性, 假设 $k < j$, 这蕴涵着

$$a(j-k) = \left(\left\lfloor \frac{aj+b}{p} \right\rfloor - \left\lfloor \frac{ak+b}{p} \right\rfloor \right) p$$

因为 $a \neq 0$ 且 $k < j$, 这蕴涵着 $a(j-k)$ 是 p 的倍数。但 $a < p$ 且 $j-k < p$, 由于 p 为素数, 因而 $a(j-k)$ 是 p 的正倍数的情况不存在 (记住每个正整数可被因式分解成素数的乘积)。因此, 如果 $j \neq k$, 则不可能存在 $f_{a,b}(j) = f_{a,b}(k)$ 的情况。换句话说, 将 Z 中的整数映射到 Z 中的整数的每个函数 $f_{a,b}$ 定义了一

个一一对应关系。因为 F 中的函数不会引起冲突,那么函数 $h_{a,b}$ 会引起冲突的唯一情况是:函数 g 会引起冲突。

126

设 j 和 k 是 Z 中两个不同的整数。另外,设 $c(j, k)$ 表示 H 中把 j 和 k 映射到同一整数(即引起 j 和 k 发生冲突)的函数个数。利用一个简单的计数论证,导出 $c(j, k)$ 的上界。如果考虑 Z 中的任一整数 x ,存在 p 个不同的函数 $f_{a,b}$,满足 $f_{a,b}(j) = x$ (因为对于选择的每个 a ,都可以选择一个 b 来满足这一条件)。现在固定 x ,注意每个这样的函数 $f_{a,b}$ 把 k 映射到 Z 中的唯一整数,且 $x \neq y$:

$$y = f_{a,b}(k)$$

此外,在 p 个形如 $y = f_{a,b}(k)$ 的不同整数中,至多存在

$$\lceil p/N \rceil - 1$$

个整数,满足 $g(y) = g(x)$,且 $x \neq y$ (根据 g 的定义)。因此,对于 Z 中的任何 x , H 中至多存在 $\lceil p/N \rceil - 1$ 个函数 $h_{a,b}$,满足

$$y = f_{a,b}(j) \text{ 且 } h_{a,b}(j) = h_{a,b}(k)$$

由于对于 Z 中的整数 x ,有 p 种选择。上述计数论证过程蕴涵着

$$c(j, k) \leq p(\lceil p/N \rceil - 1)$$

$$\leq p(p-1)/N$$

因为每个函数 $h_{a,b}$ 由满足 $0 < a < p$ 且 $0 \leq b < p$ 的数对 (a, b) 确定,因而 H 中有 $p(p-1)$ 个函数。因此,从 h 中随机均匀地选择一个函数,就是从 $p(p-1)$ 个函数中挑选一个。于是对于 Z 中任意两个不同的整数 j 和 k ,

$$\begin{aligned} \Pr(h_{a,b}(j) = h_{a,b}(k)) &\leq \frac{p(p-1)/N}{p(p-1)} \\ &= 1/N \end{aligned}$$

即族 H 是通用的。

H 中的函数除了通用性质,还有其他许多良好的性质。 H 中的每个函数易于选择,只要随机选择一对满足 $0 < a < p$ 且 $0 \leq b < p$ 的整数 a 和 b 即可。此外, H 中的每个函数易于计算。它只要求一次乘法、一次加法和应用两次模函数,因而计算时间为 $O(1)$ 。因此,从 H 中均匀随机地选择的任意散列函数都可导致字典ADT的实现,由于利用链地址规则解决冲突,所有基本操作的期望运行时间都为 $O(\lceil n/N \rceil)$ 。

127

2.6 Java 示例: 堆

为了更好地说明树ADT中的方法以及优先队列ADT对一个堆数据结构的具体实现的影响,本节讨论堆数据结构的Java实现示例。代码段2-1~2-3给出了基于堆的优先队列的Java实现。为了更具模块性,将堆结构自身的维护委托给称为堆树(heap-tree)的数据结构来完成,堆树扩展了二叉树,并另外提供了以下特殊更新方法:

- add(o)。执行下列操作序列:

```
expandExternal(z);
replaceElement(z, o);
return z;
```

在操作完成后, z 成为树中最后一个结点。

- `remove()`。执行下列操作序列：

```
t ← z.element();
removeAboveExternal(rightChild(z));
return t;
```

其中在操作开始时，`z`是最后一个结点。

即`add`操作在第一个外部结点处添加一个元素。`remove`操作删除最后一个结点上的元素。利用基于向量的树实现（见2.3.4节），操作`add`和`remove`所需时间为 $O(1)$ 。用Java接口`HeapTree`表示堆树ADT，如代码段2-1所示。假设Java类`VectorHeapTree`（未给出）用一个向量实现`HeapTree`接口，并支持运行时间为 $O(1)$ 的`add`和`remove`方法。

代码段2-1 堆树的接口`HeapTree`。它用从`PositionalContainer`接口继承而来的方法`replaceElement`和`swapElements`扩展了接口`InspectableBinaryTree`，并增加了特殊更新方法`add`和`remove`

```
public interface HeapTree extends InspectableBinaryTree, PositionalContainer {
    public Position add(Object elem);
    public Object remove();
}
```

类`HeapPriorityQueue`利用堆实现了`PriorityQueue`接口。如代码段2-2和代码段2-3所示。注意

128 把类`Item`的关键字-元素数据项存储在堆树中，类`Item`只是用于关键字-元素对的一个类。

代码段2-2 类`HeapPriorityQueue`中的实例变量、构造函数和方法`size`、`isEmpty`、`minElement`和`minKey`，它用堆实现优先队列。这个类中的其他方法如代码段2-3所示。辅助方法`key`和`element`提取优先队列一个数据项的关键字和元素，该优先队列存储在堆树中的给定位置。

```
public class HeapPriorityQueue implements PriorityQueue {
    HeapTree T;
    Comparator comp;

    public HeapPriorityQueue(Comparator c) {
        if ((comp = c) == null)
            throw new IllegalArgumentException("Null comparator passed");
        T = new VectorHeapTree();
    }

    public int size() {
        return (T.size() - 1) / 2;
    }

    public boolean isEmpty() {
        return T.size() == 1;
    }

    public Object minElement() throws PriorityQueueEmptyException {
        if (isEmpty())
            throw new PriorityQueueEmptyException("Empty Priority Queue");
        return element(T.root());
    }

    public Object minKey() throws PriorityQueueEmptyException {

```

```

        if (isEmpty())
            throw new PriorityQueueEmptyException("Empty Priority Queue");
        return key(T.root());
    }

```

```

    :

```

129

代码段2-3 类HeapPriorityQueue中的方法insertItem和removeMin。这个类中的其他方法如代码段2-2所示

```

public void insertItem(Object k, Object e) throws InvalidKeyException {
    if (!comp.isComparable(k))
        throw new InvalidKeyException("Invalid Key");
    Position z = T.add(new Item(k, e));
    Position u;
    while (!T.isRoot(z)) { // 堆向上冒泡
        u = T.parent(z);
        if (comp.isLessThanOrEqualTo(key(u), key(z)))
            break;
        T.swapElements(u, z);
        z = u;
    }
}

public Object removeMin() throws PriorityQueueEmptyException {
    if (isEmpty())
        throw new PriorityQueueEmptyException("Empty priority queue!");
    Object min = element(T.root());
    if (size() == 1)
        T.remove();
    else {
        T.replaceElement(T.root(), T.remove());
        Position r = T.root();
        while (T.isInternal(T.leftChild(r))) { // 堆向下冒泡
            Position s;
            if (T.isExternal(T.rightChild(r)) ||
                comp.isLessThanOrEqualTo(key(T.leftChild(r)), key(T.rightChild(r))))
                s = T.leftChild(r);
            else
                s = T.rightChild(r);
            if (comp.isLessThan(key(s), key(r))) {
                T.swapElements(r, s);
                r = s;
            }
        }
        break;
    }
}
return min;
}

```

130

2.7 习题

基础题

R-2.1 用伪代码描述表ADT的方法insertBefore(p, e)、insertFirst(e)和insertLast(e)的实现, 假设利用双向链

表实现表。

- R-2.2 绘制一棵表达式树，它有4个外部结点，存储数字1、5、6和7（每个外部结点存储一个数字，但未必按这个顺序）；以及3个内部结点，每个内部结点中存储二进制算术运算符 $\{+, -, \times, /\}$ 集合中的一个运算，以使根的值为21。假设运算符返回有理数（不是整数），一个运算符可能使用多次（但每个内部结点只存储一个运算符）。
- R-2.3 设 T 是一棵有多个结点有序树。 T 的前序遍历访问结点的次序与后序遍历访问结点的次序可能相同吗？如果相同，给出一个例子；否则，论证为什么这种情况不会出现。同样， T 的前序遍历访问结点的次序与后序遍历访问结点的次序可能正好相反吗？如果出现这种情况，给出一个例子；否则，论证为什么这种情况不会出现。
- R-2.4 回答下列问题，证明定理2.2。
- 绘制一棵高度为7的二叉树，使外部结点数最多。
 - 高度为 h 的二叉树中，最少外部结点数是多少？论证你的结论。
 - 高度为 h 的二叉树中，最大外部结点数是多少？论证你的结论。
 - 设 T 是高度为 h 且有 n 个结点的二叉树，证明

$$\log(n+1) - 1 \leq h \leq (n-1)/2$$

e. 上述 n 和 h 为何值时， h 的上界和下界达到相等。

- R-2.5 设 T 是一棵二叉树，满足所有外部结点有相同深度。设 D_e 是 T 中所有外部结点的深度之和， D_i 是 T 中所有内部结点的深度之和。找出常数 a 和 b ，使其满足

$$D_e + 1 = aD_i + bn$$

其中 n 是 T 中的结点数。

- R-2.6 设 T 是一棵有 n 个结点的二叉树，设 p 是 T 中结点的层编号（2.3.4节给出了其定义）。

- 证明对于 T 中每个结点 v ，有 $p(v) \leq 2^{(n+1)/2} - 1$ 。
- 给出一个至少有5个结点的二叉树的例子，满足对于某个结点 v ， $p(v)$ 的最大值达到上述上界。

- R-2.7 设 T 是一棵有 n 个结点的二叉树，用向量 S 实现。设 p 是 T 中结点的层编号（2.3.4节给出了其定义），用伪代码描述每个方法root、parent、leftChild、rightChild、isInternal、isExternal和isRoot。

- R-2.8 根据以下输入序列，说明选择排序算法的性能：

(22, 15, 36, 44, 10, 3, 9, 13, 29, 25)

- R-2.9 根据上一个问题的输入序列，说明插入排序算法的性能。

- R-2.10 对于有 n 个元素的序列的插入排序，给出一个最坏情况下的例子，并证明这样一个序列上的插入排序的运行时间为 $\Omega(n^2)$ 。

- R-2.11 堆中具有最大关键字的数据项存储在什么位置？

- R-2.12 根据以下输入序列，说明堆排序算法的性能：

(2, 5, 16, 4, 10, 23, 39, 18, 26, 15)

- R-2.13 假定用向量 S 实现一棵二叉树 T （如2.3.4节所述）。如果 n 个数据项从索引1开始，按序存储在 S 中，树 T 是一个堆吗？

- R-2.14 是否存在一个堆 T ，其中存储了7个不同的元素，满足对 T 的前序遍历会产生 T 中元素的一个有序序列。对于中序遍历和后序遍历又如何呢？

- R-2.15 证明求和 $\sum_{i=1}^n \log i$ 为 $\Omega(n \log n)$ ，该求和式出现在堆排序的分析中。

- R-2.16 显示从图2-22的堆中删除关键字16的步骤。

- R-2.17 显示从图2-22的堆中用18替代5的步骤。

- R-2.18 绘制一个堆的示例，它的关键字是1~59中的所有奇数（不重复），满足插入关键字为32的数据项的操作将会引起堆向上冒泡的过程，并经过从根的子结点向上所有的路径（用32代替子结点的关键字）。

- R-2.19 画出利用散列函数 $h(i) = (2i + 5) \bmod 11$ ，将关键字12、44、13、88、23、94、11、39、20、16和5映射到11个数据项的散列表，并假定采用链地址法解决冲突。

- R-2.20 如果采用线性探测解决冲突, 上题的结果又是什么?
- R-2.21 证明习题R-2.19的结果, 假设采用二次探测法处理冲突, 直到方法由于找不到空位而失败。
- R-2.22 习题R-2.19的结果是什么? 假设采用双散列法处理冲突, 它使用二级散列函数 $h'(k) = 7 - (k \bmod 7)$ 。
- R-2.23 给出插入散列表的伪代码描述, 它利用二次探测法解决冲突。假设利用一个特殊的“惰性数据项”对象代替被删除的数据项。
- R-2.24 对于图2-33所示的散列表, 利用再散列法, 将它变成大小为19的散列表, 使用的新散列函数为 $h(k) = 2k \bmod 19$ 。

132

创新题

- C-2.1 用伪代码描述一个链跳方法, 用于查找一个双向链表的中间结点, 该双向链表的链头和链尾为观察哨, 且有奇数个真实结点介于它们之间 (注意: 这个方法只能利用链跳; 不能利用计数器法)。这个方法的运行时间是多少?
- C-2.2 描述如何利用两个栈实现队列ADT, 使得dequeue和enqueue的平摊运行时间为 $O(1)$ 。假设栈支持方法push、pop和size的运行时间为常数。在这种情况下, enqueue()和dequeue()方法的运行时间是多少?
- C-2.3 描述如何利用两个队列实现栈ADT, 在这种情况下, push()和pop()方法的运行时间是多少?
- C-2.4 描述一个递归算法, 用于枚举数 $\{1, 2, \dots, n\}$ 的所有排列。你的方法的运行时间是多少?
- C-2.5 描述基于数组实现向量ADT的结构和伪代码, 满足在位序0处和向量末尾插入和删除的时间为 $O(1)$ 。在你的实现中, 还应该提供常量时间的方法elemAtRank。
- C-2.6 在孩子们的“热土豆”游戏中, n 个孩子围坐一圈, 沿着圈顺时针方向传递一个称为“土豆”的对象。孩子们不停传递土豆, 直到领头人按一下铃, 此时, 持有土豆的孩子必须离开游戏, 其他孩子更紧密地围坐一圈。继续这个过程, 直到圈中只剩一个孩子, 这个孩子即为获胜者。利用序列ADT, 描述一个实现这个游戏的有效方法。假定领头人总是在土豆被传递 k 次之后按铃。(确定留在这个游戏中的最后一个孩子的问题称为约瑟夫问题 (Josephus problem))。假设用双向链表实现序列, 根据 n 和 k 分析方法的运行时间; 如果用数组实现序列, 则方法的运行时间又是多少?
- C-2.7 利用序列ADT, 描述一种有效方式, 把表示 n 张牌的序列变成随机次序。利用函数randomInt(n), 它返回 $0 \sim n-1$ 之间 (包含0和 $n-1$) 的一个随机数。你的方法应该保证每种可能的次序等概率出现。如果用数组实现序列, 你的方法的运行时间是多少? 如果用链表实现序列, 你方法的运行时间又是多少?
- C-2.8 利用树遍历过程计算的量, 设计绘制二叉树的算法。
- C-2.9 设计对二叉树 T 中的结点 v 进行以下操作的算法:
- preorderNext(v): 在 T 的前序遍历中, 返回在结点 v 之后访问的结点。
 - inorderNext(v): 在 T 的中序遍历中, 返回在结点 v 之后访问的结点。
 - postorderNext(v): 在 T 的后序遍历中, 返回在结点 v 之后访问的结点。
- 你的算法在最坏情况下的运行时间是多少?
- C-2.10 给出一个 $O(n)$ 时间的算法, 计算一棵树 T 中所有结点的深度, 其中 n 是 T 中的结点数。
- C-2.11 二叉树中一个内部结点 v 的平衡因子 (balance factor) 是 v 的左、右子树高度之差。证明如何使欧拉路径遍历适用于输出一棵二叉树的所有结点的平衡因子。
- C-2.12 当且仅当下列命题之一成立, 则称两棵有序树 T' 和 T'' 是同构的 (isomorphic):
- 有序树 T' 和 T'' 均只含一个结点。
 - 有序树 T' 和 T'' 有相同数量 k 的子树, 并且 T' 的第 i 棵子树与 T'' 的第 i 棵子树同构, 对于 $i = 1, \dots, k$ 。
- 设计一个算法, 测试两棵给定的有序树是否是同构的。并分析你的算法的运行时间。
- C-2.13 令 (v, a) 数对表示欧拉路径遍历中的一次访问动作, 其中 v 是被访结点, a 是left、below和right之一。设计执行操作tourNext(v, a)的算法, 返回 (v, a) 之后的访问动作 (w, b) 。你的算法在最坏情况下的运行时间是多少?

133

- C-2.14 证明如何用一棵真二叉树表示一棵非真二叉树。
- C-2.15 设 T 是一棵有 n 个结点的二叉树。定义Roman结点 (Roman node) 是 T 中的一个结点 v , 满足 v 的左子树中的后代数量与 v 的右子树中的后代数量至多相差5。描述一个线性时间方法, 找出 T 中它自身不是Roman结点但其所有后代都是Roman结点的每个结点 v 。
- C-2.16 用伪代码描述一个非递归方法, 在一棵二叉树中进行欧拉路径遍历, 要求不用栈, 且方法运行时间为线性时间。
提示: 通过在一个结点上记录其来源于何处, 能够知道要执行哪一个访问行为。
- C-2.17 用伪代码描述一个非递归方法, 用线性时间在一棵二叉树中进行中序遍历。
- C-2.18 设 T 是一棵有 n 个结点的二叉树 (T 可用向量实现, 也可不用向量实现)。给出一个线性时间方法, 利用BinaryTree接口中的方法, 通过增加2.3.4节中给出的层编号函数 p 的值, 遍历 T 中的结点。这种遍历方法称为层序遍历 (level order traversal)。
- C-2.19 树 T 的路径长度 (path length) 是 T 中所有结点的深度之和。描述一个线性时间方法, 计算树 T 的路径长度 (T 不必是二叉树)。
- C-2.20 定义树 T 的内部路径长度 (internal path length) $I(T)$ 为 T 中所有内部结点的深度之和。同样, 定义树 T 的外部路径长度 (external path length) $E(T)$ 为 T 中所有外部结点的深度之和。证明 $E(T) = I(T) + 2n$, 其中 n 为二叉树 T 中的结点数。
- C-2.21 设 T 是一棵有 n 个结点的树。定义两个结点 v 和 w 之间的最小公共祖先LCA (lowest common ancestor) 为 T 中最低的结点, 该结点以 v 和 w 为后代 (这里允许结点是其自身的后代)。给定两个结点 v 和 w , 描述一个有效算法, 找出 v 和 w 的LCA。并分析你的算法的运行时间。
- C-2.22 设 T 是一棵有 n 个结点的树, 并且对于 T 中的任何结点 v , 设 d_v 表示 T 中结点 v 的深度。 T 中两个结点 v 和 w 之间的距离 (distance) 是 $d_v + d_w - 2d_u$, 其中 u 是 v 和 w 的LCA (定义如上题)。 T 的直径 (diameter) 是 T 中两个结点之间的最大距离。描述一个有效算法, 找出 T 的直径。并分析你的算法的运行时间。
- C-2.23 假设给定 n 个形如 $[a_i, b_i]$ 的区间集合 S 。设计一个有效算法, 计算 S 中所有区间的并集。并分析你的算法的运行时间。
- C-2.24 假设数组 A 中给出了排序问题的输入, 描述如何只利用数组 A 和另外至多6个基本类型的变量, 实现选择排序算法。
- C-2.25 假设数组 A 中给出了排序问题的输入, 描述如何只利用数组 A 和另外至多6个基本类型的变量, 实现插入排序算法。
- C-2.26 假设数组 A 中给出了排序问题的输入, 描述如何只利用数组 A 和另外至多6个基本类型的变量, 实现堆排序算法。
- C-2.27 假设用于实现堆的二叉树 T 只能利用二叉树ADT中的方法进行访问, 即不能假设将 T 实现为向量。给定对当前最后一个结点 v 的引用, 只利用二叉树接口中的方法, 描述一个有效算法找出插入点 (即新的最后一个结点)。确信考虑了所有可能的情况。并分析你的算法的运行时间。
- C-2.28 对于任意 n , 证明堆中存在插入序列, 需要 $\Omega(n \log n)$ 的时间进行处理。
- C-2.29 可以利用一个二进制串表示一棵二叉树中从根到树中某个结点的一条路径, 其中0表示“转到左子结点”, 1表示“转到右子结点”。设计一个对数时间的算法, 找出堆中最后一个结点。基于这种表示法, 堆中存放 n 个元素。
- C-2.30 证明找出堆中第 k 个最小元素的问题在最坏情况下至少需要 $\Omega(k)$ 时间。
- C-2.31 设计一个计算 n 个不同整数集中第 k 个最小元素的算法, 要求算法运行时间为 $O(n + k \log n)$ 。
- C-2.32 设 T 是存储 n 个关键字的堆。给出一个有效算法, 输出 T 中小于或等于给定查询关键字 x (它不必在 T 中) 的所有关键字。例如, 给定图2-22中的堆和一个查询关键字 $x = 7$, 算法应该输出4、5、6和7。注意关键字不一定有序输出。理想情况下, 你的算法的运行时间应为 $O(k)$, 其中 k 是输出关键字的个数。
- C-2.33 散列表字典实现要求找出介于 $M-2M$ 之间的素数。可以利用筛选算法 (sieve algorithm) 实现这个查找素数的方法。在这个算法中, 分配 $2M$ 个布尔数组 A 的单元, 满足第 i 个单元关联整数 i 。初始化数组单元为“真”, 并划出是2、3、5和7等的倍数的所有单元。这个过程一直进行, 直到大于 $\sqrt{2M}$ 的数出现为止。
- C-2.34 给出从散列表中进行一次删除的伪代码描述, 利用线性探测法解决冲突, 其中没有利用特殊标记表

示被删除的元素。也就是说，必须重排散列表的内容，使得被删除的数据项好像从未被插入在第一个位置一样。

- C-2.35 二次探测策略会产生聚集问题，这和出现冲突时它查找空位的方式有关。也就是说，当冲突在桶 $h(k)$ 中出现时，对于 $f(j) = j^2$ ，检查 $A[(h(k) + f(j)) \bmod N]$ ，其中 $i = 1, 2, \dots, N-1$ 。
- 证明对于素数 N ， $f(j) \bmod N$ 会呈现至多 $(N+1)/2$ 个不同的值，因为 j 的范围为 $1 \sim N-1$ 。作为这个论证的一部分，注意对于所有 R ， $f(R) = f(N-R)$ 。
 - 更好的策略是选择素数 N ，满足 N 与 $3 \bmod 4$ 同余，然后随着 j 从1变化至 $(N-1)/2$ ，在加和减之间交替检查桶 $A[(h(k) \pm j^2) \bmod N]$ 。证明这种交替进行的二次探测法可以保证会检查 A 中每一只桶。

程序设计

- P-2.1 编写一个程序，将一个完全加括号的算术表达式作为输入，并将它转换成为一个二进制表达式树。你的程序应该以某种方式显示这棵树，同时输出关联根结点的值。对于另一个挑战，允许叶结点存储形如 x_1 、 x_2 、 x_3 等的变量。它们初始时为0，可以由你的程序交互式更新，在表达式树的根的打印值中随之进行相应的更新。
- P-2.2 编写一个Java小程序或单机图形化程序，模拟一个堆。你的程序应该支持所有优先队列的操作，并可视化堆向上冒泡和堆向下冒泡过程的操作。（附加题：也把自底向上构造堆的过程可视化。）
- P-2.3 进行比较分析，研究各种散列编码对于字符串的冲突率，如各种多项式散列编码对于不同参数值 a 的冲突率。利用散列表确定冲突数，但仅统计将不同字符串映射到同一散列编码时的冲突数（如果映射到散列表中同一位置，则不统计）。利用在因特网上找到的文本文件测试这些散列编码。
- P-2.4 如同上一题进行比较分析，但针对10位数字的电话号码，而非字符串。

136

2.8 本章注记

本章所讨论的栈、队列和链表的基本数据结构属于计算机科学的传统内容。Knuth在他开创性的*Fundamental Algorithms* [117]一书中，首先按照时间先后给出了这些基本数据结构。在这一章里定义栈、队列和链表的基本数据结构的方法是：首先根据它们的ADT，然后再根据具体实现进行定义。由于面向对象设计方法的提出，导致软件工程在数据结构规范和实现方面取得进步。这种方法现在被认为是教授数据结构的标准方法。通过由Aho、Hopcroft和Ullman所著的关于数据结构和算法[7, 8]的经典书籍，将这种方法引入到数据结构设计中。要进一步研究抽象数据类型，请参阅Liskov和Guttag的著作[135]、Cardelli和Wegner的综述文章[44]和Demurjian的著作[57]。本章使用的关于栈、队列和出队ADT的命名约定来自JDSL[86]。JDSL是一个Java数据结构库，它建立的方法取自C++库STL[158]和LEDA[151]。对这些约定的使用将贯穿全书。在本章中，促进了栈和队列的Java实现问题的研究。对学习更多关于Java运行时环境的知识感兴趣的读者，可以参考Lindholm和Yellin著作[134]，其中定义了Java虚拟机JVM（Java Virtual Machine）。

序列和迭代器是C++标准模板库（STL）[158]中流行的概念，它们在JDSL（Java数据结构库）中起着基本性的作用。序列ADT是Java的java.util的推广和扩展。向量API（例如，见Arnold和Gosling的著作[13]）和表ADT由几位作者提出，包括Aho、Hopcroft和Ullman[8]，他们引入了“位置”抽象，并且Wood[211]定义了表ADT，类似于已定义的表ADT。Knuth开创性的*Fundamental Algorithms*[118]一书中讨论了利用数组和链表实现序列。Knuth的姊妹篇*Sorting and Searching*[119]描述了冒泡排序方法，以及该方法其他排序算法的历史。

将数据结构看作容器的概念（和其他面向对象设计的原理）可在Booch[32]和Budd[42]所著的面向对象设计书籍中找到。这些概念在Golberg和Robson[79]以及Liskov和Guttag[135]的著作中命名为“集合类”。我们使用的位置抽象源于Aho、Hopcroft和Ullman[8]引入的“位置”和“结点”

抽象。在Knuth的*Fundamental Algorithms*[118]一书中可以找到对于前序遍历、中序遍历和后序遍历方法的讨论。欧拉路径遍历技术来源于并行算法领域，由Tarjan和Vishkin[197]引入，JáJá [107]以及Karp和Ramachandran[114]对其进行了讨论。绘制一棵树的算法一般认为是绘图算法“民间传说”的一部分。对绘图有兴趣的读者可参考Tamassia[194]和Di Battista等人的著作[58、59]。习题R-2.2中的难题由Micha Sharir给出。

Knuth的关于排序和查找的著作[119]描述了选择排序、插入排序和堆排序算法的动机和历史。堆排序算法由Williams[210]提出，线性时间构造堆的算法由Floyd[70]提出。Bentley [29]、Carlsson[45]、Gonnet和Munro[82]、McDiarmid和Reed[141]以及Schaffer和Sedgewick[178]的论文中给出了其他关于堆和堆排序变体的算法和分析。定位器模式（也在[86]中描述）是新出现的。



查找树和跳跃表

人们喜欢选择。希望用不同方法解同一问题，因而可以在开销和效率之间作出平衡。本章重点研究有序字典的不同实现方法。本章首先讨论二叉查找树，然后讨论它们如何支持基于树的有序字典的实现，但是这些方法并不能保证最坏情况下的高效性能。然而，它们却构成了许多基于树的字典实现的基础。本章讨论几个这样的实现方法。其中一个经典的实现是3.2节中介绍的AVL树，它是一棵二叉查找树，查找操作和更新操作可以达到对数时间。

在3.3节中，引入了深度有界树的概念，它将所有外部结点保持在同一深度或“伪深度”。多路查找树是其中的一种，它是一棵有序树，其中每个内部结点都可以存储几个数据项且有若干个子结点。多路查找树是二叉查找树的一种推广，像二叉查找树一样，经特殊处理它能成为有序字典。3.3节讨论的一种特殊多路查找树是(2, 4)树，它是一棵深度有界查找树，其中每个内部结点分别存储1、2或3个关键字，且分别有2、3或4个子结点。这些树的优点是，插入和删除关键字的算法简单、直观。通过对树进行分裂、合并“附近”结点或在它们之间转移关键字这些自然的操作，更新操作能够重排一棵(2, 4)树。存储 n 个数据项的(2, 4)树空间复杂度为 $O(n)$ ；在最坏情况下，查找、插入和删除操作的运行时间为 $O(\log n)$ 。本节研究的另一种深度有界树是红黑树。在这些二叉树中，结点按照一定方式被着色以“红色”和“黑色”。这种着色模式保证每个外部结点位于同一“黑色深度”（对数级）。黑色深度的“伪深度”表示源于说明红黑树和(2, 4)树之间的对应关系。利用这种对应关系，提出了在红黑树中进行插入和删除的更复杂一些的算法，这些方法基于旋转和重新着色操作。红黑树相比其他二叉查找树（如AVL树）所具有的优势是在插入和删除之后，只用 $O(1)$ 时间的旋转重构。

在3.4节中，讨论了伸展树，它的吸引力在于查找和更新方法的简洁性。伸展树是二叉查找树，每次查找、插入或删除之后，通过仔细设计旋转序列，将被访问的结点向上移动到根。这种简单“移动到顶”的启发式策略帮助数据结构自身适合于(adapt)所进行的操作。启发式策略的结果之一是，伸展树能够保证每个字典操作的平摊运行时间为对数时间。

最后，在3.5节中讨论了跳跃表，它不是树数据结构，但它具有深度概念，并把所有元素保持在对数深度。这些结构是随机的，因而它们的深度界限是概率意义上的。这就是在一个存储 n 个元素的跳跃表中，其高度以非常高的概率达到 $O(\log n)$ 。这肯定不是真正的最坏情况下的界限，但是跳跃表的更新操作相当简单，且实际上人们喜欢将它们比喻为查找树。

给定AVL树和红黑树的Java实现，集中讨论3.6节二叉查找树的实现。强调如何用这两个数据结构构建2.3节讨论的树ADT。

本章毫无疑问讨论了相当多的查找结构, 由于读者或教师时间有限, 可能只对其中的一些主题感兴趣。由于这个原因, 在设计这一章时, 除了3.1节之外, 其余各节与其他节都是独立的。

3.1 有序字典和二叉查找树

在有序字典中, 希望进行通常的字典操作, 这些操作在2.5.1节讨论过。这些操作有 $\text{findElement}(k)$ 、 $\text{insertItem}(k, e)$ 和 $\text{removeElement}(k)$, 同时希望在字典中维持关键字的一个有序关系。可以利用比较器提供关键字之间的顺序关系。正如将要看到的那样, 这样的顺序有助于高效地实现字典ADT。此外, 一个有序字典还支持以下方法:

- $\text{closestKeyBefore}(k)$: 返回小于或等于 k 、具有最大关键字的数据项的关键字。
- $\text{closestElemBefore}(k)$: 返回小于或等于 k 、具有最大关键字的数据项的元素。
- $\text{closestKeyAfter}(k)$: 返回大于或等于 k 、具有最小关键字的数据项的关键字。
- $\text{closestElemAfter}(k)$: 返回大于或等于 k 、具有最小关键字的数据项的元素。

如果字典中不存在满足查询条件的数据项, 上述每个方法都会返回一个特殊 NO_SUCH_KEY 对象。

上述操作的有序性质利用了日志文件或者散列表, 它们不适合于实现字典, 这是因为这些数据结构都不会维持字典中关键字的任何顺序信息。的确, 当关键字几乎随机分布在散列表中时, 散列表得到它们最佳的查找速度。因此, 当处理有序字典时, 应该考虑新的字典实现方法。

[141]

现在已经定义了字典抽象数据类型, 现在看看实现这个ADT的一些简单方式。

3.1.1 有序表

如果一个字典 D 是有序的, 可以把它的数据项按照关键字的非降序存储在向量 S 中。这里指定 S 是向量, 而不是一般的序列。因为向量 S 中关键字有序, 可以进行快速查找, 如果 S 为链表则不然。称一个字典 D 的有序向量实现为查找表 (lookup table)。我们来比较一下这个实现与利用无序序列实现字典的日志文件。

查找表所需空间复杂度类似于日志文件, 为 $\Theta(n)$ 。假设可以对支持向量 S 的数组进行增长和收缩, 使数组大小与 S 中的数据项数成正比。但是, 不像日志文件, 在查找表中进行更新所需时间相当长。特别是在查找表中进行 $\text{insertItem}(k, e)$ 操作最坏情况下需要的时间为 $O(n)$, 因为需要上移向量中具有大于 k 的关键字的所有数据项, 以便腾出存放新数据项 (k, e) 的空间。根据最坏情况下字典更新操作的运行时间, 查找表实现次于日志文件。然而, 我们可以在查找表中更快地执行 findElement 操作。

二分搜索

利用基于数组的向量 S 实现 n 个数据项的有序字典 D 的重大好处是, 通过 S 中元素的位序 (rank) 访问元素所需时间为 $O(1)$ 。由2.2.1节可知, 向量中元素的位序就是该元素之前的元素个数。因此, S 中第一个元素的位序为0, 最后一个元素的位序为 $n-1$ 。

S 中的元素是字典中的数据项, 由于 S 是有序的, 位序为 i 的数据项的关键字不小于位序为 $0, \dots, i-1$ 的数据项的关键字, 且不大于位序为 $i+1, \dots, n$ 的数据项的关键字。这个观察结果使我们可以利用孩子们的“高一低”游戏的一个变体, 快速查找关键字 k 。如果在当前查找阶段不能排除 I 的关键字等于 k , 称 D 中的数据项 I 为候选项 (candidate)。算法中有两个参数, low 和 high , 满足所有候选数据项的位序至少为 low , 至多为 high 。初始时, $\text{low} = 0$ 和 $\text{high} = n-1$, 令 $\text{key}(i)$ 表示位序为 i 的关键字, $\text{elem}(i)$ 为它的元素。然后, 将 k 与中值候选项的关键字进行比较, 即与位序为

$$\text{mid} = \lfloor (\text{low} + \text{high}) / 2 \rfloor$$

的数据项比较。

考虑下面三种情况：

- 如果 $k = \text{key}(\text{mid})$ ，那么找到所要的数据项，查找成功终止，并返回 $\text{elem}(\text{mid})$ 。
- 如果 $k < \text{key}(\text{mid})$ ，那么递归调用向量的前半部分，即范围为 $\text{low} \sim \text{mid} - 1$ 。
- 如果 $k > \text{key}(\text{mid})$ ，那么递归调用向量的后半部分，即范围为 $\text{mid} + 1 \sim \text{high}$ 。

142

这种查找方法称为二分搜索 (binary search)，算法3-1给出了它的伪代码。在基于向量实现的 n 个数据项的字典上，操作 $\text{findElement}(k)$ 由调用 $\text{BinarySearch}(S, k, 0, n-1)$ 组成。

算法3-1 有序向量中的二分搜索

算法 $\text{BinarySearch}(S, k, \text{low}, \text{high})$:

输入：有序向量 S ，存放 n 个数据项，通过方法 $\text{key}(i)$ 访问它的关键字，通过方法 $\text{elem}(i)$ 访问它的元素；查找关键字 k ；以及整数 low 和 high

输出：如果 S 中存在关键字为 k 的元素，且其位序介于 low 和 high 之间，则输出该元素；否则，输出特殊元素 NO_SUCH_KEY

if $\text{low} > \text{high}$ then

 return NO_SUCH_KEY

else

$\text{mid} \leftarrow \lfloor (\text{low} + \text{high}) / 2 \rfloor$

 if $k = \text{key}(\text{mid})$ then

 return $\text{elem}(\text{mid})$

 else if $k < \text{key}(\text{mid})$ then

 return $\text{BinarySearch}(S, k, \text{low}, \text{mid} - 1)$

 else

 return $\text{BinarySearch}(S, k, \text{mid} + 1, \text{high})$

图3-1说明了二分搜索算法。

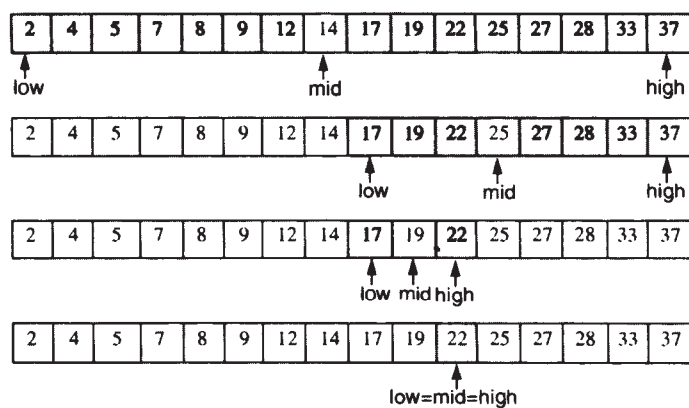


图3-1 在具有整型关键字的字典中执行操作 $\text{findElement}(22)$ 的二分搜索示例，该字典是用基于数组的有序向量实现的。为简单起见，只显示了存放在字典中的关键字，而没有显示元素

143

考虑二分搜索的运行时间，观察可知每次递归调用需要执行常数次操作。因此，运行时间与递归调用次数成正比。一个重要的事实是，对于每次递归调用，需要在 S 中查找的候选数据项的个数是由值 $\text{high} - \text{low} + 1$ 给出的，而且每次递归调用，所剩候选项的个数至少会减半。确切地讲，

依据mid的定义, 所剩候选项的个数为

$$(mid-1) - low + 1 = \lfloor (low + high)/2 \rfloor - low \leq (high - low + 1)/2$$

或

$$high - (mid+1) + 1 = high - \lfloor (low + high)/2 \rfloor \leq (high - low + 1)/2$$

起初候选项数是 n ; 第一次调用BinarySearch之后, 候选项个数至多为 $n/2$; 第二次调用BinarySearch之后, 候选项个数至多为 $n/4$; 依此类推。也就是说, 如果令函数 $T(n)$ 表示此方法的运行时间, 那么可以把递归二分搜索算法的运行时间表征如下:

$$T(n) \leq \begin{cases} b & \text{如果 } n < 2 \\ T(n/2) + b & \text{否则} \end{cases}$$

其中 b 是常数。一般而言, 这个递归方程表明, 每次递归调用之后, 剩余候选数据项的个数至多为 $n/2^i$ 。(5.2.1节将会详细讨论这种形式的递归方程。)在最坏情况下(查找不成功), 没有更多的候选数据项时, 递归调用停止。因此, 所进行的递归调用的最大次数就是满足 $n/2^m < 1$ 的最小整数 m 。换句话说(请注意, 对数底数为2时省略了底数), $m > \log n$ 。因此, 可得 $m = \lfloor \log n \rfloor + 1$, 这蕴涵着BinarySearch($S, k, 0, n-1$)的运行时间为 $O(\log n)$ 。

表3-1比较了日志文件或查找表实现的字典方法的运行时间。日志文件可以进行快速插入, 但是查找和删除操作较慢; 而查找表可以进行快速查找, 但是插入和删除操作较慢。

表3-1 日志文件和有序表实现的有序字典的主要方法的运行时间比较。 n 表示方法执行时字典中的数据项数。方法closestElemBefore、closestKeyAfter、closestElemAfter的性能类似于closestKeyBefore

方 法	日志文件	查找表
findElement	$O(n)$	$O(\log n)$
insertItem	$O(1)$	$O(n)$
removeElement	$O(n)$	$O(n)$
closestKeyBefore	$O(n)$	$O(\log n)$

3.1.2 二叉查找树

本节讨论的二叉查找树这种数据结构将二分搜索过程的思想应用于基于树的数据结构。定义二叉查找树是一棵二叉树, 其中每个内部结点 v 存放一个元素 e , 满足存储在 v 的左子树中的元素小于或等于 e , 存储在 v 的右子树中的元素大于或等于 e 。此外, 假设外部结点不存储元素; 因此, 它们事实上可以为空, 或引用NULL_NODE对象。

二叉查找树的中序遍历按照非降序访问存储在这样一棵树中的元素。二叉查找树支持进行查找, 它在查找每个内部结点时会比较该结点上的元素是小于、等于, 还是大于正被查找的元素。

可以利用一棵二叉查找树 T , 通过向下遍历树 T , 定位一个给定值为 x 的元素。在每个内部结点上, 比较当前结点与查找元素 x 的值。如果结果为“小于”, 则继续在左子树中进行查找; 如果结果为“等于”, 则查找成功终止; 如果结果为“大于”, 则继续在右子树中进行查找。最后如果到达一个外部结点(为空), 那么查找不成功终止。如图3-2所示。

3.1.3 二叉查找树中的查找

形式上, 二叉查找树(binary search tree)是一棵二叉树 T , 其中 T 的每个内部结点 v 存储字典

D 中的一个数据项 (k, e) , 存储在 v 的左子树中的结点上的关键字小于或等于 k , 存储在 v 的右子树中的结点上的关键字大于或等于 k 。

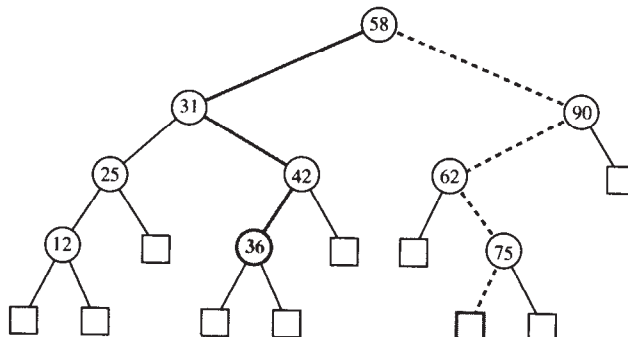


图3-2 存储整数的二叉查找树。用粗线画出的粗实线路径是成功查找36时的遍历路径。粗虚线路径是不成功查找70时的遍历路径

在算法3-2中，基于上述二叉查找树中的查找策略，给出了一个递归方法TreeSearch。给定查找关键字 k 和 T 中的一个结点 v ，方法TreeSearch返回 T 的子树 $T(v)$ （以 v 为根）的一个结点（位置） w ，满足出现以下两种情况之一：

- w 是 $T(v)$ 的一个存储关键字 k 的内部结点。
- w 是 $T(v)$ 的一个外部结点，在中序遍历中， $T(v)$ 中 w 之前的所有内部结点的关键字小于 k ； w 之后的所有内部结点的关键字则大于 k 。

因此，可通过调用 T 上的方法TreeSearch($k, T.root()$)在字典 D 上执行方法findElement(k)。令 w 是这个TreeSearch方法调用返回的 T 的结点。如果结点 w 是内部结点，则返回 w 中存储的元素；否则，如果 w 是外部结点，那么返回NO_SUCH_KEY。

算法3-2 二叉查找树中的递归查找

算法 TreeSearch(k, v):

输入：查找关键字 k ，二叉查找树 T 中的一个结点 v

输出：根为 v 的 T 的子树 $T(v)$ 中的一个结点 w ，满足 w 要么是一个存储关键字 k 的内部结点；要么是一个外部结点，其中数据项（如果它存在）的关键字为 k

if v 是外部结点 then

return v

if $k = \text{key}(v)$ then

return v

else if $k < \text{key}(v)$ then

return TreeSearch($k, T.\text{leftChild}(v)$)

else

{已知 $k > \text{key}(v)$ }

return TreeSearch($k, T.\text{rightChild}(v)$)

注意：二叉查找树 T 中查找的运行时间与树 T 的高度成正比。因为树有 n 个结点，树的高度小则为 $O(\log n)$ ，大则为 $\Omega(n)$ ，因此当二叉查找树树高较小时，其查找效率最高。

二叉查找树分析

在一棵二叉查找树 T 中进行查找，其最坏情况下运行时间的形式分析是简单的。对于它所遍

历的树中的每个结点，二分搜索算法会执行常数个基本操作。在前一个结点的子结点上进行新一步的遍历。也就是说，二叉树查找算法是从根向下每次一层沿树中的结点执行的。因此，这样的结点个数被限定为 $h+1$ ，其中 h 为 T 的高度。换句话说，因为会在查找中遇到的每个结点上花费 $O(1)$ 时间，方法findElement（或任何其他标准查找操作）的运行时间为 $O(h)$ ，其中 h 是用于实现字典 D 的二叉查找树 T 的高度，如图3-3所示。

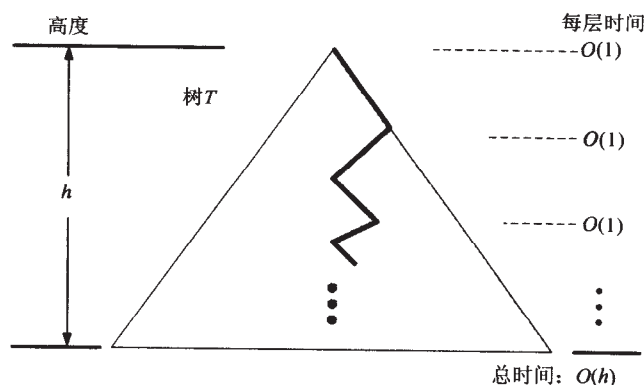


图3-3 二叉查找树中查找的运行时间说明。图中利用标准的可视化方式，将二叉查找树看作一个大三角形和从根开始的Z字形路径

也可显示上述算法的一个变体，它能够执行操作findAllElements(k)，在 $O(h + s)$ 时间内，找到字典中关键字为 k 的所有数据项。然而，这个方法稍微复杂一些，详细的过程留作习题（C-3.3）。

当然， T 的高度 h 可能与 n 一样大，但通常希望它更小一些。在本章后续几节中，将表明如何维持把查找树 T 的高度的一个上界 $O(\log n)$ 。不过，在描述这个模式之前，首先描述在可能不平衡的二叉查找树中字典更新方法的实现。

147

3.1.4 二叉查找树中的插入

可以利用相当直接（但是有价值）的算法，实现二叉查找树中的insertItem操作和removeElement操作。

对于基于二叉查找树实现的字典 D ，为了在其上执行insertItem(k, e)操作，开始时，在 T 上调用方法TreeSearch($k, T.root()$)。设 w 是TreeSearch返回的结点。

- 如果 w 是一个外部结点（ T 中不存在关键字为 k 的数据项），通过 T 上的操作expandExternal(w)（见2.3.3节），用一个存储数据项(k, e)的新内部结点和两个外部子结点代替 w 。注意 w 是插入关键字为 k 的数据项的合适位置。
- 如果 w 是一个内部结点（关键字为 k 的另一数据项存储在 w 中），调用TreeSearch($k, rightChild(w)$)（或者等价地调用TreeSearch($k, leftChild(w)$)），并递归将算法应用于TreeSearch返回的结点。

上述插入算法最终描绘出一条从 T 的根向下至外部结点的路径，该外部结点将由包含新数据项的一个新内部结点替代。因此，插入将新数据项添加到查找树 T 的“底部”。图3-4显示了向二叉查找树中插入数据项的过程。

插入算法的分析类似查找的分析。最坏情况下所访问的结点数与 T 的高度 h 成正比。同时假设用链表结构实现 T （见2.3.4节），访问每个结点的时间为 $O(1)$ 。因此，方法insertItem的运行时间为 $O(h)$ 。

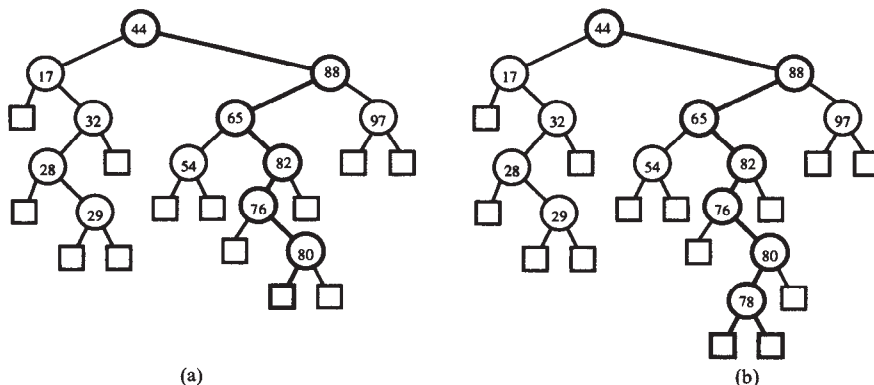


图3-4 将关键字为78的数据项插入二叉查找树中。(a)中显示找到了插入位置；(b)中显示结果树

148

3.1.5 二叉查找树中的删除

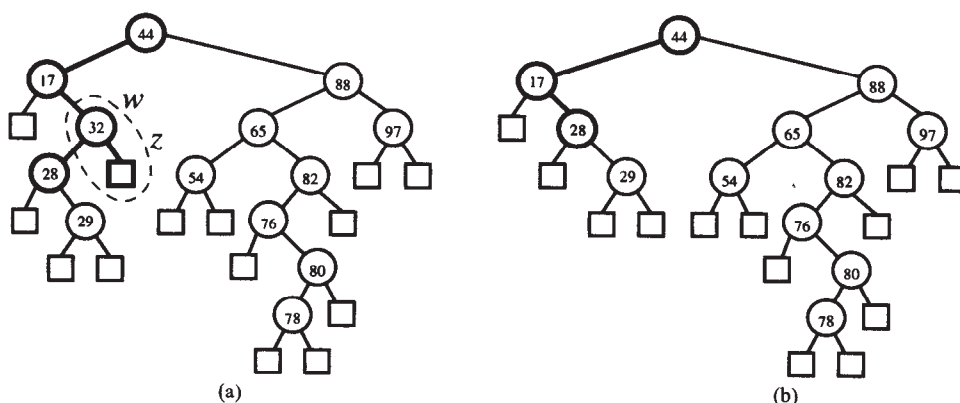
在二叉查找树实现的字典 D 上进行 $\text{removeElement}(k)$ 操作有点复杂,因为不希望在树 T 中创建任何“洞”。这样一个洞是一个未存放元素的内部结点,它使得在二叉查找树中进行正确查找变得困难(如果不是不可能的话)。如果进行许多删除操作,而不重构树 T ,那么就会有大量内部结点没有存放任何元素,这会使将来的查找变得混乱。

因为通过在 T 上执行算法 $\text{TreeSearch}(k, T.\text{root}())$ 查找存储关键字 k 的结点开始删除操作,因此删除操作开始时足够简单。如果 TreeSearch 返回一个外部结点,那么在字典 D 中不存在关键字为 k 的元素,则返回特殊元素 NO_SUCH_KEY 。如果 TreeSearch 返回一个内部结点 w ,那么 w 中存放有希望删除的数据项。

基于 w 是否是一个容易删除的结点,将处理过程分成两种情形(难度增加):

- 如果结点 w 的其中一个子结点 z 是一个外部结点,则可以执行操作 $\text{removeAboveExternal}(z)$,从 T 中简单地删除 w 和 z 。这个操作(见图2-15和2.3.4节)用 z 的兄弟结点代替 w 来重构 T ,并同时从 T 中删除结点 w 和 z 。

这种情况如图3-5所示。

图3-5 以图3-4b为例,进行二叉查找树中的删除,其中要删除的关键字是32,它存储在结点 w 中, w 有一个外部结点:(a)删除之前的树 T ,以及 T 上受到 $\text{removeAboveExternal}(z)$ 操作影响的结点;(b)删除之后的树 T

149

- 如果 w 的两个子结点都是内部结点,则不能简单地从 T 中删除结点 w ,因为这将会在 T 中形

成一个“洞”。而是进行如下处理（如图3-6所示）：

- (1) 按中序遍历树 T ，找到 w 后的第一个内部结点 y 。结点 y 是 w 的右子树中最左边的内部结点，它是在首次访问 w 的右子结点时找到的结点。然后从那儿沿 T 向下，接着访问左子结点。同时， y 的左子结点 x 是外部结点，在 T 的中序遍历中，它紧跟在结点 w 之后。
- (2) 将 w 中的元素存放在临时变量 t 中，并将 y 的数据项移到 w 中。这个动作会删除 w 中以前存放的数据项。
- (3) 利用操作`removeAboveExternal(x)`，从 T 中删除 x 和 y 。这个动作用 x 的兄弟结点代替 y ，并从 T 中删除 x 和 y 。
- (4) 返回以前存储在 w 中的元素，已经把它存放在临时变量 t 中。

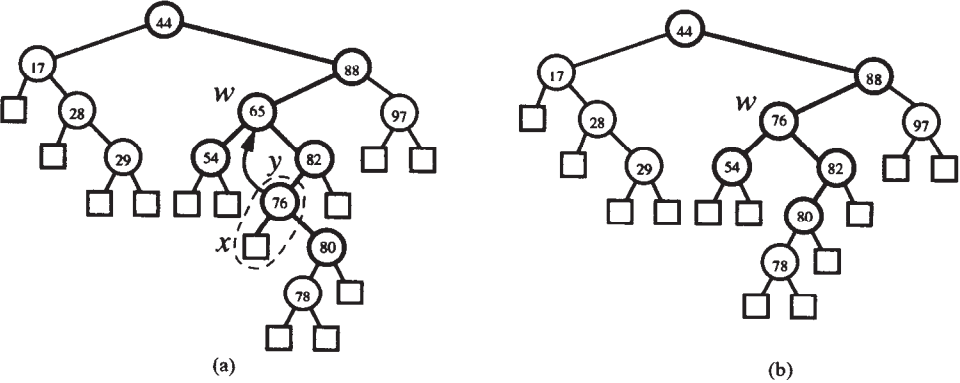


图3-6 以图3-4b为例，进行二叉查找树中的删除，其中要删除的关键字是65，它存储在有两个内部子结点的结点中：(a)删除之前；(b)删除之后

删除算法的分析类似于插入算法和查找算法的分析。访问每个结点所需的时间为 $O(1)$ ，最坏情况下所访问的结点数与 T 的高度 h 成正比。因此，对于二叉查找树实现的字典 D ，方法`removeElement`的运行时间为 $O(h)$ ，其中 h 为树的高度。

150

还可证明，上述算法的一个变体可以在 $O(h + s)$ 时间内执行操作`removeAllElements(k)`，其中 s 是返回的迭代器中的元素个数。详细过程留作习题（C-3.4）。

3.1.6 二叉查找树的性能

二叉查找树实现的字典 D 的性能概括在定理3.1和表3-2中。

定理3.1 对于有 n 个关键字-元素的数据项、高度为 h 的二叉查找树 T ，利用 $O(n)$ 的空间，执行字典ADT操作的运行时间如下。操作`size`和`isEmpty`的运行时间均为 $O(1)$ 。操作`findElement`、`insertItem`和`removeElement`的运行时间均为 $O(h)$ 。操作`findAllElements`和`removeAllElements`的运行时间均为 $O(h + s)$ ，其中 s 是返回的迭代器的大小。

表3-2 二叉查找树实现的字典中主要方法的运行时间。 h 表示当前树的高度， s 是`findAllElements`和`removeAllElements`返回的迭代器的大小。所用空间为 $O(n)$ ，其中 n 为存储在字典中的数据项数

方 法	时 间
<code>size</code> 、 <code>isEmpty</code>	$O(1)$
<code>findElement</code> 、 <code>insertItem</code> 、 <code>removeElement</code>	$O(h)$
<code>findAllElements</code> 、 <code>removeAllElements</code>	$O(h + s)$

注意,在一棵二叉查找树中查找和更新操作的运行时间随着树高的不同而变化很大。然而平均来讲,通过插入和删除关键字的随机序列产生的 n 个关键字的二叉查找树的预期高度为 $O(\log n)$ 。这样一个声明需用严谨的数学语言精确定义随机插入和删除序列的意义,以及用复杂的概率论知识进行证明。因而,它的证明超出了本书的范围。因此,随机更新序列导致二叉查找树平均具有对数高度是令人满意的结果,但是记住它们最坏情况下的低劣性能。在将标准的二叉查找树用于更新不是随机操作的应用中时需要小心。

二叉查找树的相对简单性及其良好的平均性能使得二叉查找树在某些应用中成为相当有吸引力的字典数据结构,在这些应用中,插入和删除的关键字遵循一种随机模式,因而偶尔有些慢的响应时间也是可以接受的。然而,对于某些应用,最坏情况下的快速查找时间和更新时间至关重要。下一节介绍的数据结构将会满足这种需求。

151

3.2 AVL 树

在上一节中,讨论了有效的字典数据结构应该是什么,但是对于某些操作,它所获得的最坏情况下的性能为线性时间,这并不比基于序列的字典实现(如日志文件和查找表)的性能更好。在本节中将描述这个问题的一种简单的修正方法,使之对于所有基本的字典操作获得对数时间。

定义

简单修正是在二叉查找树定义中添加一条规则,该规则维持树的一个对数高度。本节考虑的规则是如下的高度平衡性质(height-balance property),它根据其内部结点的高度,表征二叉查找树 T 的结构(回忆2.3.2节,树中结点 v 的高度就是从 v 到一个外部结点的最长路径长度):

高度平衡性质:对于 T 中的每个内部结点 v , v 的子结点的高度至多相差1。

称任何一棵满足这个性质的二叉查找树 T 为AVL树,它是以其发明者名字的首字母缩写词命名的一个概念,其发明者为Adel'son-Vel'skii和Landis。AVL树的一个例子如图3-7所示。

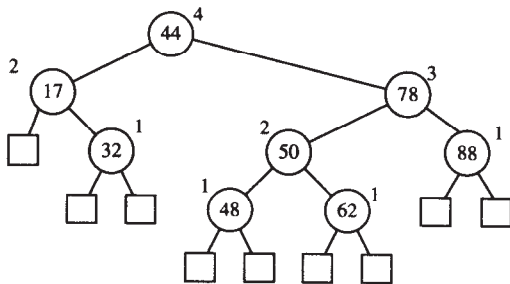


图3-7 AVL树的一个例子。结点内为关键字,高度则显示在结点旁边

由高度平衡性质可知,一棵AVL树的子树自身也是一棵AVL树。高度平衡性质的另一个重要结论是,保持高度较小,如以下命题所示。

152

定理3.2 一棵存储 n 个数据项的AVL树高度为 $O(\log n)$ 。

证明 我们不是试图直接找出一棵AVL树高的上界,一种更容易的方式是解决它的“逆问题”,找出一棵具有最少内部结点数 $n(h)$ 的AVL树,其高度为 h 。证明 $n(h)$ 至少呈指数增长,即 $n(h)$ 为 $\Omega(c^h)$,其中 $c > 1$ 为常数。由此,容易导出存储 n 个关键字的AVL树高度为 $O(\log n)$ 。

注意, $n(1) = 1$ 且 $n(2) = 2$,因为高度为1的AVL树至少有一个内部结点,高度为2的AVL树至少有两个内部结点。对于 $h \geq 3$,高度为 h 且结点数最少的AVL树,满足其两棵子树都是结点数最

少的AVL树, 其中一棵高度为 $h-1$, 另一棵高度为 $h-2$ 。考虑加上根, 得到以下将 $n(h)$ 关联到 $n(h-1)$ 和 $n(h-2)$ 的公式, 对于 $h \geq 3$:

$$n(h) = 1 + n(h-1) + n(h-2) \quad (3.1)$$

公式(3.1)蕴涵 $n(h)$ 是 h 的严格递增函数 (对应于斐波那契数列)。因此, 得到 $n(h-1) > n(h-2)$ 。在公式(3.1)中, 用 $n(h-2)$ 代替 $n(h-1)$, 并去掉1得, 对于 $h \geq 3$:

$$n(h) > 2n(h-2) \quad (3.2)$$

公式(3.2)表明每当 h 增加2, $n(h)$ 至少加倍, 这表明 $n(h)$ 呈指数增长。为了形式化证明这个事实, 反复应用公式(3.2), 用简单的归纳论证, 证明

$$n(h) > 2^i n(h-2i) \quad (3.3)$$

对于任何整数 i , 满足 $h-2i \geq 1$ 。因为已经知道 $n(1)$ 和 $n(2)$ 的值, 选择 i , 使之满足 $h-2i$ 等于1或2, 即选择 $i = \lceil h/2 \rceil - 1$ 。把上述 i 值代入公式(3.3)中, 得

$$\begin{aligned} n(h) &> 2^{\lceil h/2 \rceil - 1} n(h - 2\lceil h/2 \rceil + 2) \\ &\geq 2^{\lceil h/2 \rceil - 1} n(1) \\ &\geq 2^{\lceil h/2 \rceil - 1} \end{aligned} \quad (3.4)$$

对公式(3.4)两端取对数, 得 $\log n(h) > h/2 - 1$, 由此可得

$$h < 2\log n(h) + 2 \quad (3.5)$$

这蕴涵着一棵存储 n 个关键字的AVL树的高度至多为 $2\log n(h) + 2$ 。 ■

153

由定理3.2及3.1.2节给出的二叉查找树的分析可知, 在用AVL树实现的字典中进行findElement操作的运行时间为 $O(\log n)$, 其中 n 是字典中的数据项数。

3.2.1 更新操作

余下的重要问题是说明当执行插入或删除之后, 如何维持一棵AVL树的高度平衡性质。AVL树的插入和删除操作类似于二叉查找树中的操作, 但对于AVL树必须增加一些计算。

1. 插入

AVL树 T 中的插入操作开始于3.1.4节中描述的 (简单) 二叉查找树中的insertItem操作。回忆这个操作总是向 T 中以前是一个外部结点的新结点 w 中插入一个新数据项。通过操作expandExternal使 w 成为内部结点, 即给 w 添加两个外部子结点。然而对于某些高度增加1的结点, 这个动作可能违反高度平衡性质。特别是结点 w 以及它的某些可能的祖先使其高度增加1。因此, 下面描述如何重构 T 以恢复其高度平衡性质。

给定二叉查找树 T , 对于 T 中的一个结点 v , 如果其子结点之间的高度之差的绝对值至多为1, 则称结点 v 是平衡的 (balanced); 否则称为不平衡的 (unbalanced)。因此, 表征AVL树的高度平衡性质等价于说每个内部结点是平衡的。

假定 T 满足高度平衡性质, 因此在插入新的数据项之前, 它是一棵AVL树。在 T 上进行操作expandExternal(w)之后, T 中某些结点 (包括 w) 的高度增加, 所有这样的结点都在 T 中从 w 到根的路径上。而且这些可能是变为不平衡的唯一结点。如图3-8a所示。当然, 如果这种情况发生, 那么 T 不再是一棵AVL树; 因此, 需要一种机制修正所造成的“不平衡”。

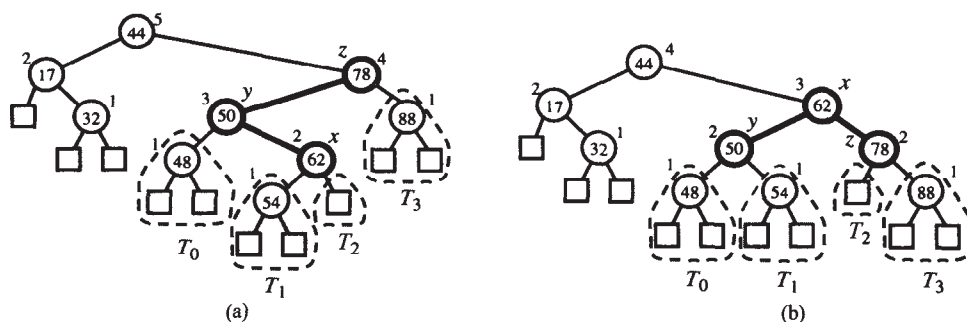


图3-8 将关键字为54的元素插入到图3-7中的AVL树中的例子：(a)为关键字54加入新结点之后，存储关键字78和44的结点变成不平衡结点；(b)三结点重构恢复高度平衡性质。显示它们相邻结点的高度，并用 x 、 y 和 z 标识它们

154

通过简单的查找-修正策略，恢复二叉查找树 T 中结点的平衡。特别地，在从 w 向树根查找的过程中，设 z 是遇见的第一个不平衡结点。如图3-8a所示。令 y 表示 z 的具有更高高度的子结点（注意 y 一定是 w 的祖先）。令 x 表示 y 的具有更高高度的子结点（如果存在一个tie，选择 x 为 w 的祖先），注意结点 x 可以等于 w ，且 x 是 z 的孙子结点。由于在以其子结点 y 为根的子树中执行插入操作，导致 z 变得不平衡，因此 y 的高度比其兄弟结点的高度大2。通过调用算法3-3中描述的二结点重构（trinode restructuring）方法restructure(x)，重新平衡以 z 为根的子树。如图3-8和图3-9所示。三结点重构暂时将结点 x 、 y 和 z 重命名为 a 、 b 和 c ，以使得在中序遍历中， a 在 b 之前， b 在 c 之前。有四种将结点 x 、 y 和 z 映射为 a 、 b 和 c 的可能方式，如图3-9所示。通过重新标记，它们被统一为一种情况。三结点重构用称为 b 的结点代替 z ，并使该结点的子结点为 a 和 c ，使 a 和 c 的子结点为以前 x 、 y 和 z 的四个子结点（除了 x 和 y ），与此同时保持 T 中所有结点的中序关系。

算法3-3 二叉查找树中的三结点重构操作

算法 restructure(x):

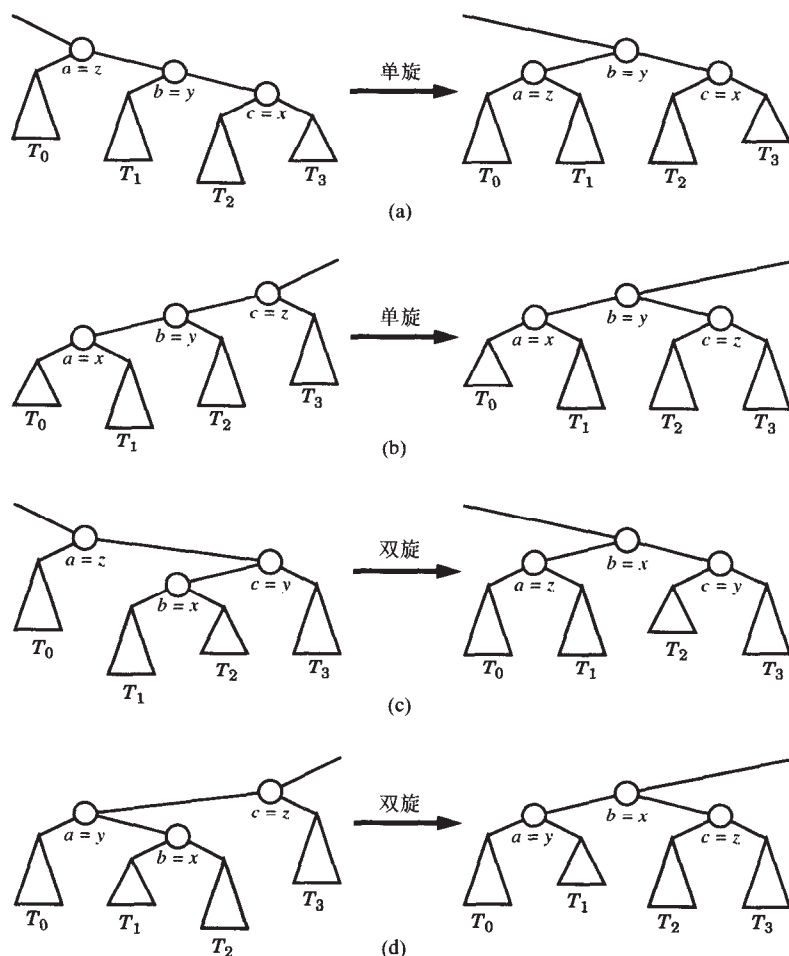
输入：二叉查找树 T 中的一个结点 x ，它有一个父结点 y 和一个祖父结点 z

输出：三结点重构之后的树 T （对应于一次或两次旋转），包含 x 、 y 和 z

1. 设 (a, b, c) 是结点 x 、 y 和 z 从左到右（中序）的列表， (T_0, T_1, T_2, T_3) 是 x 、 y 和 z 的四棵子树从左到右（中序）的列表，且这些子树不以 x 、 y 和 z 为根。
2. 用根为 b 的新子树代替根为 z 的子树。
3. 设 a 是 b 的左子结点， T_0 和 T_1 分别是 a 的左、右子树。
4. 设 c 是 b 的右子结点， T_2 和 T_3 分别是 c 的左、右子树。

常常称三结点重构导致树 T 的修正为旋转（rotation），这是由于可以用几何方式来可视化它更改 T 的方式。如果 $b = y$ （见算法3-3），称三结点重构方法为单旋（single rotation），因为可将其看作 y 沿 z “旋转”。如图3-9a和图3-9b所示。如果 $b = x$ ，称三结点重构为双旋（double rotation），因为首先可将其看作 x 沿 y “旋转”，然后再沿 z 旋转。如图3-9c和图3-9d、图3-8所示。有些研究人员把这两种情况看作不同的方法，每个都有两种对称形式；但是，已经选择统一了这四种旋转的类型。无论从哪个角度查看它，三结点重构方法都用 $O(1)$ 时间修正 T 中父-子关系的结点，同时保持 T 中所有结点的中序遍历次序。

155



156

图3-9 三结点重构操作的示意图说明 (算法3-3)。(a)和(b)部分显示单旋; (c)和(d)部分显示双旋

三结点重构操作除了保序性外,还改变了 T 中结点的高度以恢复平衡。回想一下由于 x 的祖父 z 不平衡,而执行方法 $\text{restructure}(x)$ 。然而,这种不平衡是由于 x 的一个子结点现在的高度相对于 z 的其他子结点的高度显得过大。旋转的结果把 x 的“高”子结点向上移动,同时把 x 的“低”子结点向下移动。因此,执行 $\text{restructure}(x)$ 之后,以 b 为根的子树中的所有结点是平衡的。如图3-9所示。因此,在结点 x 、 y 和 z 处,局部 (locally) 恢复了高度平衡性质。

此外,由于进行新数据项的插入之后,以 b 为根的子树代替了以前以 z 为根的子树 (该子树要高一个单位),以前不平衡的 z 的所有祖先结点变为平衡。如图3-8所示 (这一事实的证明留作习题C-3.13)。因此,这个重构也全局 (globally) 恢复了平衡性质。也就是说,在一棵AVL树中进行一次插入之后,一次旋转 (单旋或双旋) 就足以恢复高度平衡性质。

2. 删除

正如 insertItem 字典操作,首先利用普通二叉查找树上进行操作的算法,实现一棵AVL树 T 上的 removeElement 字典操作。在一棵AVL树中利用这一方法带来的困难是,它可能违反高度平衡性质。

特别地,在执行 $\text{removeAboveExternal}$ 操作,删除一个内部结点,并将它的一个子结点提升到它所在的位置之后,从先前被删除结点的父结点 w 到 T 的根的路径上,可能在 T 中存在不平衡的结

点。如图3-10a所示。事实上，至多有一个这样的不平衡结点（这个事实的证明留作习题C-3.12）。

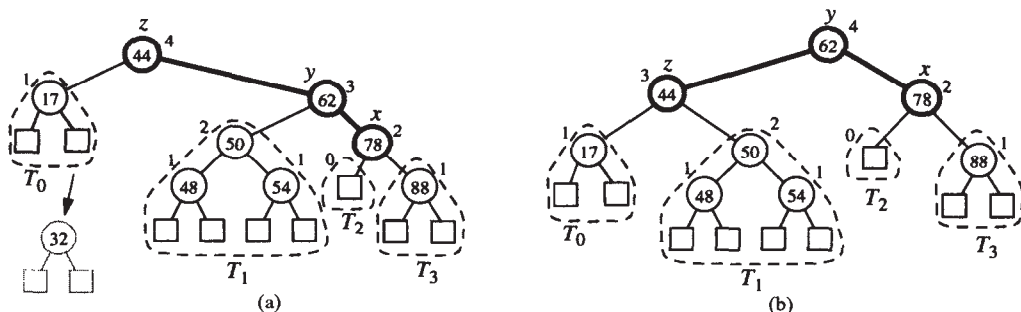


图3-10 从图3-7的AVL树中，删除关键字为32的元素：(a)删除存储关键字32的结点之后，根结点变成不平衡结点；(b)一次单旋恢复高度平衡性质

157

至于插入，利用三结点重构恢复树 T 的平衡性质。特别是在从 w 向树的根的查找过程中，设 z 是遇见的第一个不平衡结点。设 y 表示 z 的具有更高高度的子结点（注意 y 是 z 的子结点，且 z 不是 w 的祖先）。令 x 是 y 的子结点，定义如下：如果 y 的一个子结点比另一个高，则设 x 是较高的子结点；否则（ y 的两个子结点具有相同高度），设 x 是与 y 在同一边的那个结点（也就是说，如果 y 是一个左子结点，则令 x 是 y 的左子结点，否则设 x 是 y 的右子结点）。在任何情况下，当进行 $\text{restructure}(x)$ 操作时，它局部恢复了子树高度平衡性质，这棵树以前以 z 为根，现在以暂时称为 b 的结点为根，如图3-10b所示。

三结点重构可能将以 b 为根子树的高度减1，这反过来又会导致 b 的祖先成为不平衡结点。因此，执行删除之后，一次三结点重构也许不能全局恢复高度平衡性质。因此，重新平衡 z 之后，继续沿 T 向上寻找不平衡结点。如果找到另一个不平衡结点，则进行重构操作恢复它的平衡性质，并继续沿 T 向上寻找更多的不平衡结点，一直到树根。因为树 T 的高度依然为 $O(\log n)$ ，其中 n 是数据项数，由定理3.2可知， $O(\log n)$ 时间的三结点重构足以恢复高度平衡性质。

3.2.2 性能

下面总结了对AVL树的分析。操作 findElement 、 insertItem 和 removeElement 沿着 T 从根到叶的路径访问结点，可能再加上它们的兄弟结点，每个结点所需时间为 $O(1)$ 。因此，由定理3.2可知树 T 的高度为 $O(\log n)$ ，则每个上述操作的运行时间为 $O(\log n)$ 。图3-11说明了这一性能。

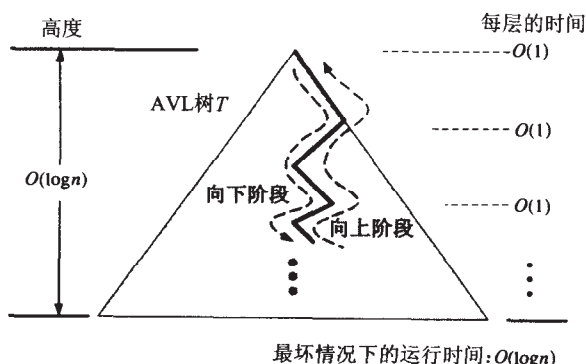


图3-11 AVL树中查找和更新的运行时间说明。每层所需时间为 $O(1)$ ，分为向下阶段和向上阶段，前者通常涉及查找，后者通常涉及更新高度值以及进行局部三结点重构（旋转）

158

3.3 深度有界查找树

某些查找树的效率与其明确限定的深度有关。事实上，这些树一般定义一个深度函数，或与深度关系密切的“伪深度”函数，使得每个外部结点位于相同的深度或伪深度。因为这样做使得在一棵有 n 个元素的树中能够保持每个外部结点的深度为 $O(\log n)$ 。由于树的查找时间和更新时间通常与深度成正比，因此这样的深度有界树（depth-bounded tree）可用于实现一个有序字典，它具有 $O(\log n)$ 时间的查找和更新。

3.3.1 多路查找树

某些深度有界查找树是多路树，即树中的内部结点有两个或更多的子结点。本节描述如何用多路树作为查找树，包括多路树如何存储数据项，以及如何在多路查找树中进行查找操作。回忆查找树中存储的数据项是形如 (k, x) 的数对，其中 k 是关键字， x 是与关键字相关的元素。

设 v 是有序树的一个结点，如果 v 有 d 个子结点，则称 v 是一个 d 结点。定义多路查找树（multi-way search tree）是一棵有序树 T ，它具有如下性质（如图3-12a中的说明）：

- T 的每个内部结点至少有两个子结点，即每个内部结点都是一个 d 结点，其中 $d \geq 2$ 。
- T 的每个内部结点存储形如 (k, x) 的数据项集合，其中 k 是关键字， x 是元素。
- T 的每个 d 结点 v 的子结点为 v_1, \dots, v_d ，存储 $d-1$ 个数据项 $(k_1, x_1), \dots, (k_{d-1}, x_{d-1})$ ，其中 $k_1 \leq \dots \leq k_{d-1}$ 。
- 定义 $k_0 = -\infty$ 和 $k_d = +\infty$ 。对于存储在以 v_i 为根的 v 的子树中的每个数据项 (k, x) ，有 $k_{i-1} \leq k \leq k_i$ ，其中 $i = 1, \dots, d$ 。

也就是说，如果把存储在 v 中的关键字集合看作包含虚构的特殊关键字 $k_0 = -\infty$ 和 $k_d = +\infty$ ，那么存储在 T 的以子结点 v_i 为根的子树中的关键字 k 一定位于存储在 v 中的两个关键字“之间”。这个简单的观点导致一个具有 d 个子结点的结点存储 $d-1$ 个常规关键字，同时形成多路查找树中查找算法的基础。

由上述定义可知，多路查找树的外部结点并不存储任何数据项，而只作为“占位符”。因此，可将一棵二叉查找树（3.12节）看作多路查找树的一个特例。在另一种极端情况下，一棵多路查找树只有一个内部结点，它存储所有数据项。此外，在外部结点可以为空（null）时，可以简化假设，把这些外部结点看作不存储任何数据项的实际结点。

但是，不论一棵多路查找树的内部结点有两个孩子还是多个孩子，在数据项数与外部结点数之间都存在有趣的关系。

定理3.3 存储 n 个数据项的多路查找树有 $n+1$ 个外部结点。

该定理的证明留作习题（C-3.16）。

1. 多路树中的查找

给定一棵多路查找树 T ，查找关键字为 k 的元素是简单的。从 T 的根开始沿一条路径进行查找。如图3-12b和图3-12c所示。在查找中如果处于一个 d 结点 v 处，则将关键字 k 与存储在 v 处的关键字 k_1, \dots, k_{d-1} 进行比较。如果对于某些 i ，有 $k = k_i$ ，则查找成功完成。否则，继续查找满足 $k_{i-1} < k < k_i$ 的 v 的子结点 v_i （回想一下曾经定义 $k_0 = -\infty$ 和 $k_d = +\infty$ ）。如果查找到达一个外部结点，那么可知 T 中不存在关键字为 k 的数据项，查找不成功终止。

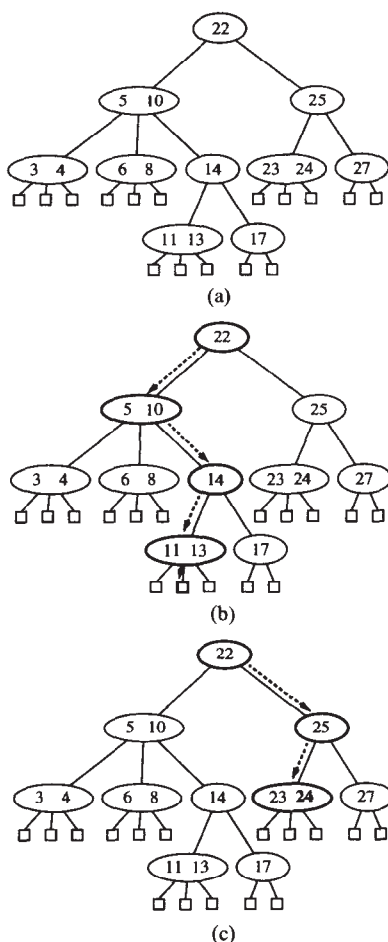


图3-12 (a)一棵多路查找树 T ; (b)在 T 中查找关键字12的查找路径(不成功的查找); (c)在 T 中查找关键字24的查找路径(成功的查找)

2. 多路查找树的数据结构

在2.3.4节中,讨论了表示一般树的不同方法。这些表示中的每一种都可用于多路查找树。事实上,在利用一般多路树实现多路查找树时,需要存储在每个结点上的唯一附加信息是关联那个结点的数据项集(包括关键字)。也就是说,需要在 v 中存储指向某些容器或者对象集合(它们存储 v 的数据项)的一个引用。

回想一下当用一棵二叉树表示一个有序字典 D 时,只需简单地在每个内部结点上存储一个指向单一数据项的引用。在利用多路查找树 T 表示 D 时,必须在 T 的每个内部结点 v 中存储一个指向关联 v 的有序数据项集的引用。这种推理初看上去像是一个循环证明,因为需要一个有序字典的表示去表示一个有序字典。但是,可以利用自展(bootstrapping)技术,避免循环证明。在这种技术中,利用问题的前一个解建立一个新(更高级)的解。在这种情况下,自展技术包含利用前面构造的字典数据结构(例如,基于有序向量的查找表,如3.1.1节所示),来表示关联每个内部结点的有序集。特别地,假设已有一种实现有序字典的方式,能够取一棵树 T ,并在 T 中每个 d 结点 v 中存储这样一个字典,从而实现一棵多路查找树。

称存储在每个结点 v 中的字典为二级(secondary)数据结构,因为要用它支持更大的主

(primary) 数据结构。用 $D(v)$ 表示存储在 T 的结点 v 中的字典。在查找操作过程中, 存储在 $D(v)$ 中的数据项允许找出将哪个子结点移到下一个结点。确切地讲, 对于 T 中的每个结点 v , 其子结点为 v_1, \dots, v_d , 数据项为 $(k_1, x_1), \dots, (k_{d-1}, x_{d-1})$ 。把数据项 $(k_1, x_1, v_1), (k_2, x_2, v_2), \dots, (k_{d-1}, x_{d-1}, v_{d-1}), (+\infty, \text{null}, v_d)$ 存储在字典 $D(v)$ 中。也就是说, 字典 $D(v)$ 中的一个数据项 (k_i, x_i, v_i) 的关键字为 k_i , 元素为 (x_i, v_i) 。注意最后一项存储特殊关键字 $+\infty$ 。

有了上述多路查找树 T 的实现, 当在 T 中查找关键字为 k 的元素时, 将执行一个处理 d 结点 v 的操作, 其方法是在 $D(v)$ 中查找具有大于或等于 k 的最小关键字的数据项 (k_i, x_i, v_i) 。如同 $\text{closestElemAfter}(k)$ 操作 (见 3.1 节)。这分两种情况:

- 如果 $k < k_i$, 那么通过处理子结点 v_i 继续查找 (注意: 如果返回特殊关键字 $k_d = +\infty$, 那么 k 大于存储在结点 v 中的所有关键字, 通过处理子结点 v_d 继续查找)。
- 否则 ($k = k_i$), 那么查找成功终止。

3. 多路查找树的性能问题

考虑上述一棵多路查找树 T 的实现的空间需求, 假设 T 中存储 n 个数据项, 由定理 3.3 可知, 利用任何一种有序字典的通用实现 (2.5 节) 来实现树 T 结点的二级结构, T 所需的总空间为 $O(n)$ 。

下面考虑回应 T 中的一次查找所需的时间。在查找过程中, 在 T 的 d 结点 v 上的所花费的时间取决于二级数据结构 $D(v)$ 的实现。如果用基于向量的有序序列 (如查找表) 实现 $D(v)$, 那么结点 v 上的处理时间为 $O(\log d)$ 。如果利用无序序列 (如日志文件) 实现 $D(v)$, 那么结点 v 上的处理时间为 $O(d)$ 。设 d_{\max} 表示 T 中任意结点的最大子结点数, h 表示 T 的高度, 在一棵多路查找树中的查找时间为 $O(hd_{\max})$ 或 $O(h \log d_{\max})$, 这取决于 T 中结点的二级数据结构的特定实现 (字典 $D(v)$)。如果 d_{\max} 为常数, 则进行一次查找所需时间为 $O(h)$, 而与二级结构的实现无关。

因此, 多路查找树的主要效率目标是保持高度尽可能小, 即希望 h 是字典中存储的数据项总数 n 的对数函数。称一棵具有对数高度的查找树为平衡查找树 (balanced search tree)。深度有界查找树通过把所有外部结点正好保持在树中具有相同深度的层上, 来满足这个目标。

接下来将讨论一棵深度有界树, 它是一棵将 d_{\max} 限定到 4 的多路查找树。在 14.1.2 节中, 将讨论更一般的多路查找树及其某些应用, 其中的查找树是如此之大, 以至于不能完全适合于计算机的内存。

3.3.2 (2,4)树

在实际中利用多路查找树时, 希望它是平衡的, 即具有对数高度。下面研究的多路查找树相当容易保持平衡。它是一棵 (2,4) 树, 有时也称为 2-4 树, 或者 2-3-4 树。事实上, 可以通过维持两个简单的性质, 来维持一棵 (2,4) 树的平衡 (如图 3-13 所示):

- 大小性质 (size property): 每个结点至多有 4 个子结点。
- 深度性质 (depth property): 所有外部结点的深度相同。

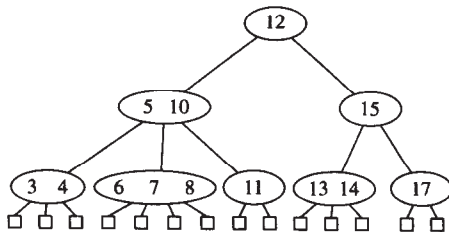


图3-13 一棵(2,4)树

维持(2, 4)树的大小性质, 可以保持多路查找树中结点的大小为常数, 因此, 可用常数大小的数组表示存储在每个内部结点 v 中的字典 $D(v)$ 。另一方面, 深度性质通过强制一棵树成为深度有界结构, 维持(2, 4)树的平衡。

定理3.4 存储 n 个数据项的一棵(2, 4)树的高度为 $\Theta(\log n)$ 。

证明 设 h 是存储 n 个数据项的一棵(2, 4)树的高度。注意, 由大小性质可知, 深度1上至多有4个结点, 深度2上至多有 4^2 个结点, 依此类推。因此, T 中的外部结点个数至多为 4^h 。同样, 由(2, 4)树的深度性质和定义可知, 深度1上至少有2个结点, 深度2上至少有 2^2 个结点, 依此类推。因此, T 中的外部结点个数至少为 2^h 。此外, 由定理3.3可知, T 中外结点个数为 $n+1$ 。于是, 有

$$2^h \leq n+1 \text{ 且 } n+1 \leq 4^h$$

对上述每一项取以2为底的对数, 则得

$$h \leq \log(n+1) \text{ 且 } \log(n+1) \leq 2h$$

定理得证。 ■ 163

1. (2, 4)树中的插入

定理3.4表明, 大小性质和深度性质足以保持一棵多路查找树是平衡的。但是, 在一棵(2, 4)树中进行插入和删除之后, 维持这些性质需要做一些工作。尤其是, 为了把关键字为 k 的新数据项 (k, x) 插入一棵(2, 4)树 T 中, 首先进行对 k 的查找。假设 T 中不存在关键字为 k 的元素, 这次查找将不成功地终止于外部结点 z 。设 v 是 z 的父结点, 把新数据项插入在结点 v 处, 并给 v 添加一个新的子结点 w (一个外部结点), 它在 z 的左边, 即向字典 $D(v)$ 中添加一个数据项 (k, x, w) 。

因为在与现有外部结点相同的层上添加一个新的外部结点, 因此插入方法保持了深度性质。但是, 它可能违反大小性质。的确, 如果一个结点 v 以前是一个4-结点, 那么插入之后, 可能变成5-结点, 这使得树 T 不再是一棵(2, 4)树。称这种违反大小性质的情形为结点 v 处的上溢(overflow)。因此, 必须解决它以恢复一棵(2, 4)树的性质。设 v_1, \dots, v_5 是 v 的子结点, k_1, \dots, k_4 是存储在 v 中的关键字。为了修正结点 v 处的上溢, 在结点 v 处进行如下分裂(split)操作, 如图3-14所示:

- 用两个结点 v' 和 v'' 代替 v , 其中
 - v' 是一个3-结点, 它的子结点为 v_1 、 v_2 和 v_3 , 并存储关键字 k_1 和 k_2 。
 - v'' 是一个2-结点, 它的子结点为 v_4 和 v_5 , 并存储关键字 k_4 。
- 如果 v 是 T 的根, 建立一个新的根结点 u ; 否则, 设 u 是 v 的父结点。
- 将关键字 k_3 插入 u 中, 使 v' 和 v'' 成为 u 的子结点, 满足如果 v 是 u 的子结点 i , 那么 v' 和 v'' 分别成为 u 的子结点 i 和 $i+1$ 。

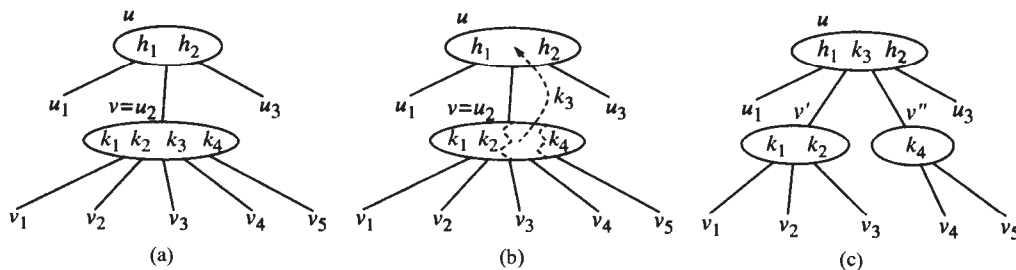


图3-14 结点分裂: (a)在5-结点 v 处上溢; (b) v 的第三个关键字被插入 v 的父结点 u 中; (c)用3-结点 v' 和2-结点 v'' 代替结点 v

图3-15显示了在(2, 4)树中的一系列插入操作。

分裂操作会影响树中的常数个结点。这些结点中存储了 $O(1)$ 个数据项。因此,可实现这个操作的运行时间为 $O(1)$ 。

164

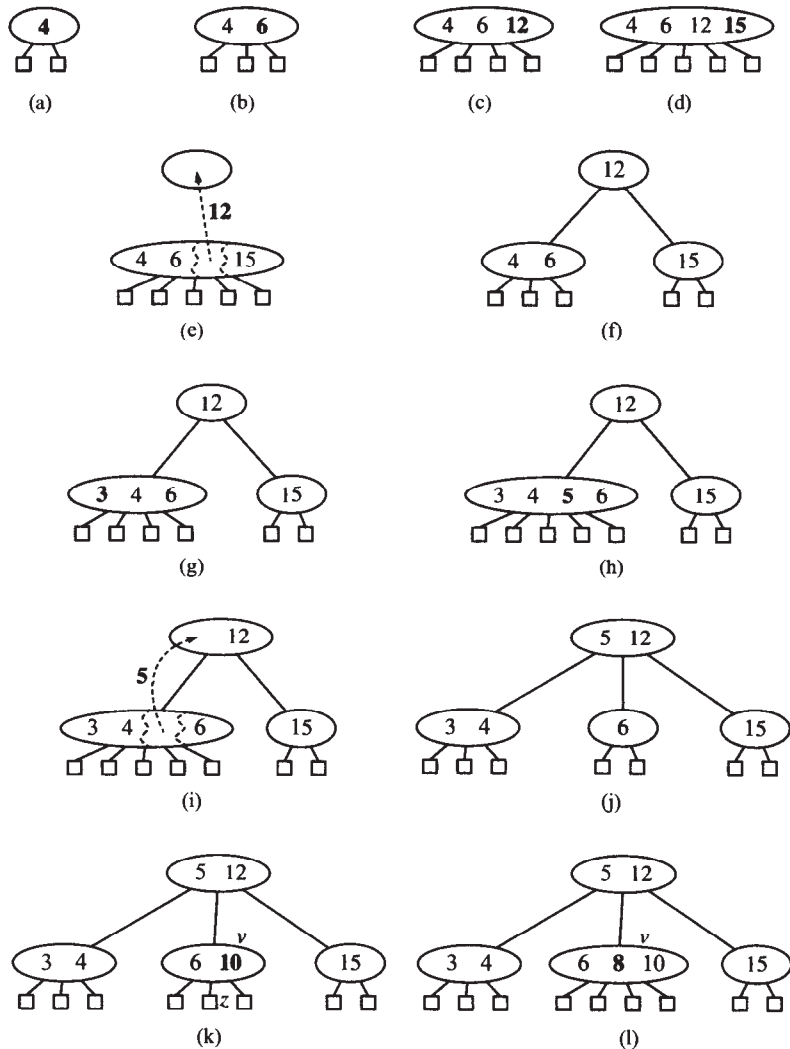


图3-15 在 $(2, 4)$ 树中的一系列插入操作: (a)具有一个数据项的初始树; (b)插入6; (c)插入12; (d)插入15, 引起上溢; (e)分裂, 建立新的根结点; (f)分裂后; (g)插入3; (h)插入5, 引起上溢; (i)分裂; (j)分裂后; (k)插入10; (l)插入8

165

2. $(2, 4)$ 树插入的性能

结点 v 上的分裂操作完成后, 在 v 的父结点 u 处可能出现新的上溢。如果出现这样的上溢, 它反过来会触发结点 u 上的分裂。如图3-16所示。分裂操作要么消除上溢, 要么将其传播到当前结点的父结点中。实际上, 这种传播可能沿着所有方向继续, 直到向上到达查找树的根。但是, 如果它确实沿着所有方向一直传播到根, 则它最终会在那个结点得到解决。图3-16显示了这样一个分裂传播序列。

因此, 分裂操作数受限于树的高度, 由定理3.4可知, 高度为 $O(\log n)$ 。于是, 在 $(2, 4)$ 中进行一次插入操作的总时间为 $O(\log n)$ 。

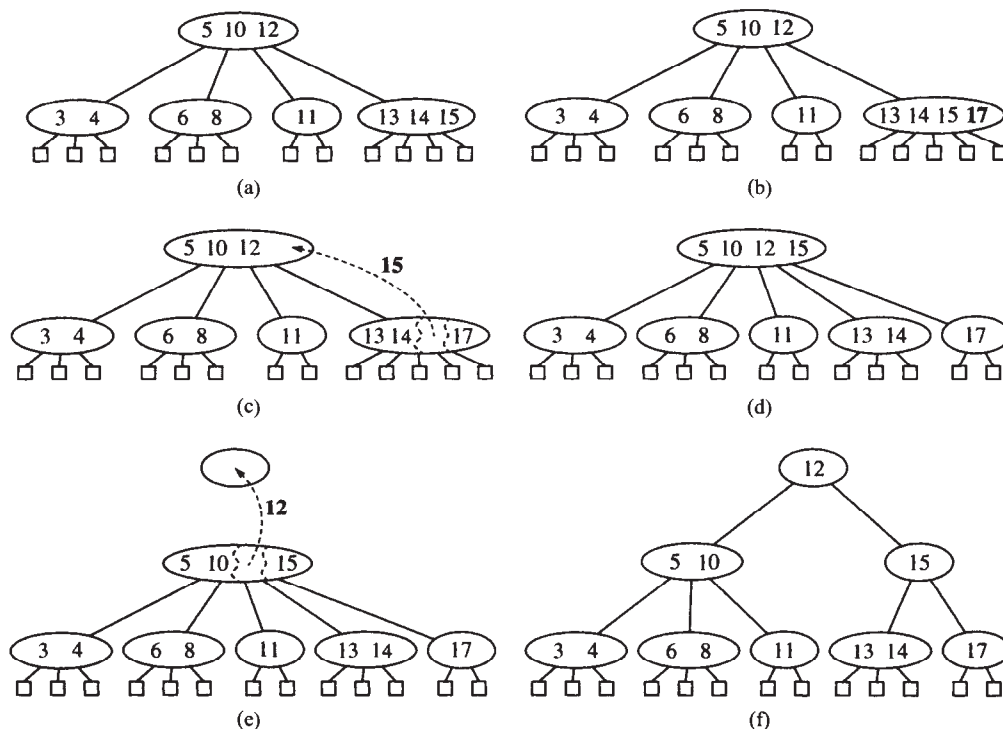


图3-16 在(2,4)树中进行一次插入导致层叠分裂: (a)插入之前; (b)插入17, 引起上溢; (c)分裂; (d)分裂之后, 出现新的上溢; (e)另一次分裂, 建立新的根结点; (f)最终树

166

3. (2,4)树中的删除

现在考虑在一棵(2,4)树 T 中删除关键字为 k 的数据项。首先在 T 中查找关键字为 k 的数据项。对于在(2,4)树中删除这样一个数据项, 总是可将其归约到这样一种情况, 即被删除的数据项存储在其子结点是外部结点的结点 v 中。例如, 假定希望删除关键字为 k 的数据项, 它存储在只有内部子结点的结点 z 的第 i 个数据项(k_i, x_i)中。在这种情况下, 将(k_i, x_i)与存储在具有外部子结点的结点 v 中的相应数据项交换, 其操作方式如下。如图3-17d所示。

- (1) 在以 z 的第 i 个子结点为根的子树中找到最右边的内部结点 v 。
- (2) 将 z 的数据项(k_i, x_i)与 v 的最后一个数据项交换。

一旦确定要删除的数据项存储在只有外部子结点的结点 v 中(因为它要么已经在 v 中, 要么将它交换到 v 中), 简单地从 v 中(即从字典 $D(v)$ 中)删除该数据项, 并删除 v 的第 i 个外部结点。

上面描述的从结点 v 中删除一个数据项(和一个子结点)保持了深度性质, 因为总是从只有外部子结点的结点 v 中删除一个外部子结点。然而, 在删除这样一个外部结点时, 可能违反 v 处的大小性质。如果 v 以前是一个2-结点, 那么删除之后, 它成为一个没有数据项的1-结点。如图3-17d和图3-17e所示。称这种大小性质违反为结点 v 处的下溢(underflow)。为了修正下溢, 检查 v 的一个直接近邻兄弟结点是一个3-结点, 还是一个4-结点。如果找到这样一个兄弟结点 w , 那么进行转移(transfer)操作, 将 w 的一个子结点转移到 v 中, 将 w 的关键字转移到 v 和 w 的父结点 u 中, 并将 u 的关键字转移到 v 中。如图3-17b和图3-17c所示。如果 v 只有一个兄弟结点, 或者如果 v 的两个直接近邻兄弟结点是2-结点, 那么进行合并(fusion)操作, 将 v 与兄弟结点合并, 建立新结点 v' , 并从 v 和 v' 的父结点 u 中移动一个关键字。如图3-17e和图3-17f所示。

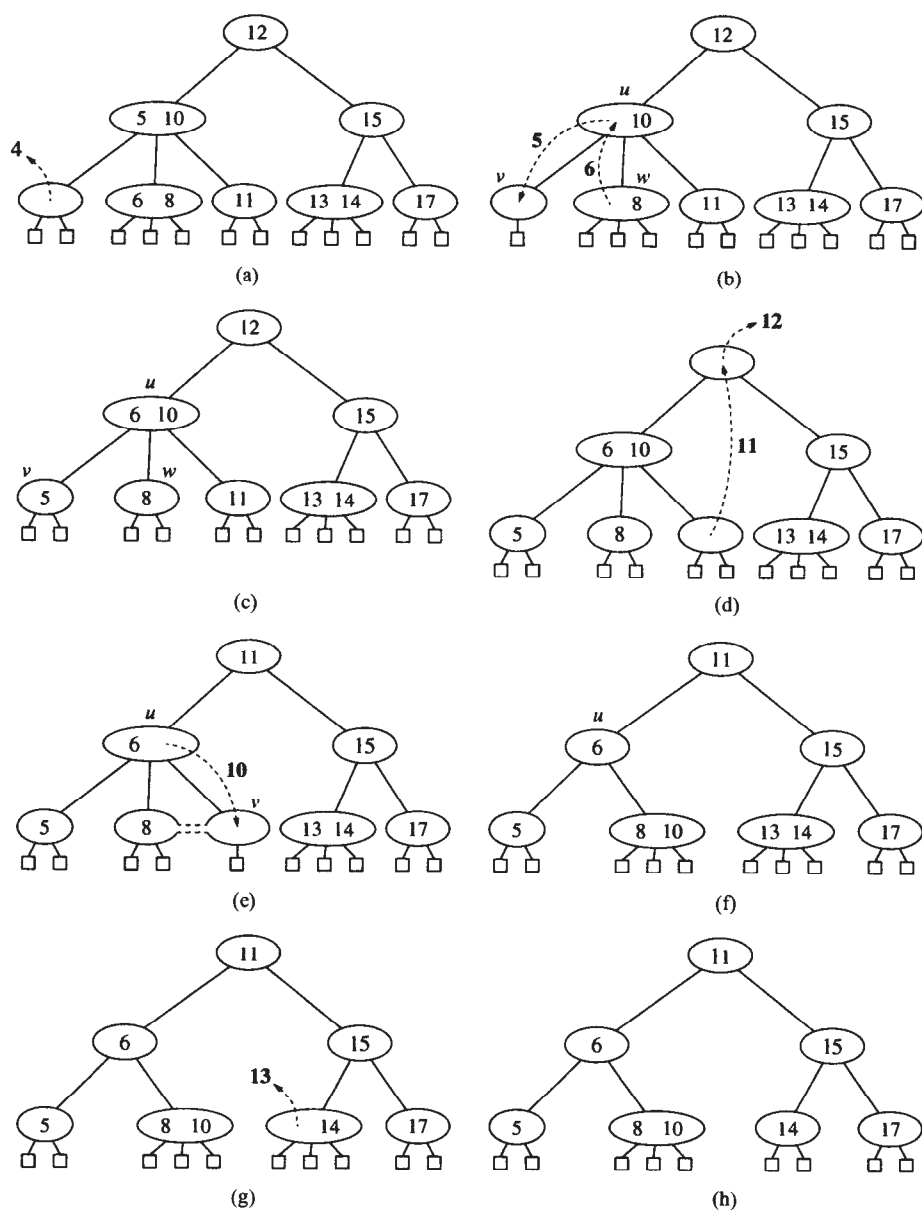


图3-17 一棵(2,4)树中的一系列删除操作: (a)删除4, 引起下溢; (b)转移操作; (c)转移操作之后; (d)删除12, 引起下溢; (e)合并操作; (f)合并操作之后; (g)删除13; (h)删除13之后

结点 v 处的合并操作可能在 v 的父结点 u 处引起新的下溢, 这反过来又触发 u 处的转移或合并。如图3-18所示。因此, 合并操作数受限于树的高度, 由定理3.4可知, 高度为 $O(\log n)$ 。如果下溢沿所有方向一直向上传播到根, 那么根就会被删除。如图3-18c和图3-18d所示。图3-17和图3-18显示了一棵(2,4)树中的一系列删除操作。

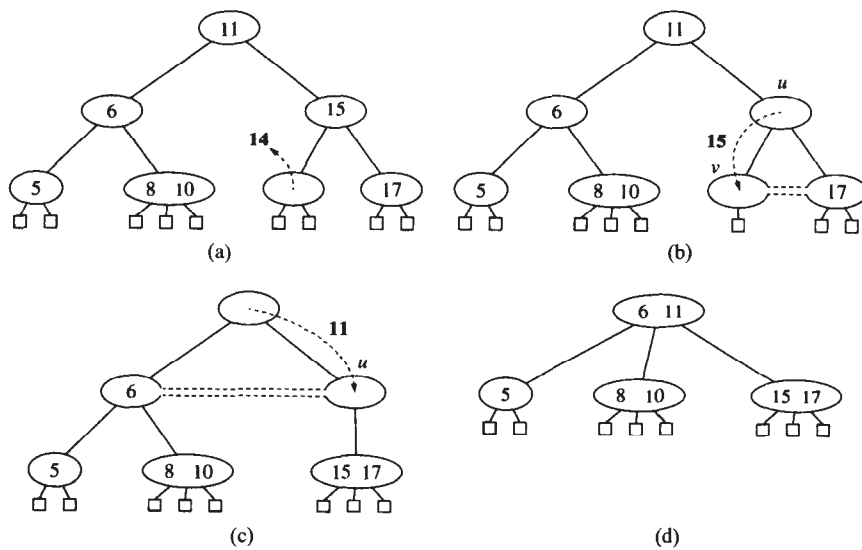


图3-18 一棵(2,4)树中合并的传播序列：(a)删除14，引起下溢；(b)合并，引起另一次下溢；(c)第二次合并操作，删除根；(d)最终树

4. (2,4)树的性能

表3-3总结了用(2,4)树实现字典的主要操作的运行时间。时间复杂度的分析基于如下事实：

- 存储 n 个数据项的(2,4)树的高度为 $O(\log n)$ （根据定理3.4）。
- 一次分裂、转移或合并操作的时间为 $O(1)$ 。
- 一次查找、插入或删除数据项的操作要访问 $O(\log n)$ 个结点。

表3-3 (2,4)树实现的有 n 个元素的字典的性能，其中 s 表示findAllElements操作和removeAllElements操作返回的迭代器大小。所用空间为 $O(n)$

操 作	时 间
size、isEmpty	$O(1)$
findElement、insertItem、removeElement	$O(\log n)$
findAllElements、removeAllElements	$O(\log n + s)$

因此(2,4)树提供对字典的快速查找和更新操作。(2,4)树还与下面要讨论的数据结构之间具有一种有趣的关系。

169

3.3.3 红黑树

本节讨论的数据结构（红黑树）是一种二叉查找树，它通过深度有界查找树的方法，利用一种“伪深度”达到平衡。特别地，红黑树（red-black tree）是一棵二叉查找树，其结点按照一定方式着红色和黑色，它满足以下性质：

- 根性质：根着黑色。
- 外部性质：每个外部结点着黑色。
- 内部性质：一个红色结点的子结点着黑色。
- 深度性质：所有外部结点有相同的黑色深度（black depth），定义为黑色祖先数减1。

红黑树的一个示例如图3-19所示。在本节中，约定黑色结点（black node）及其父结点的边缘用粗线（thick line）绘制。

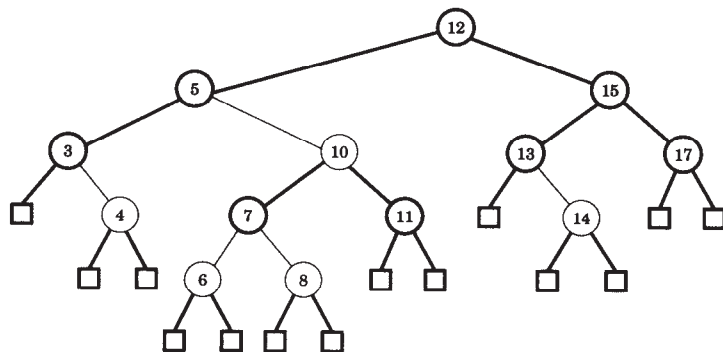


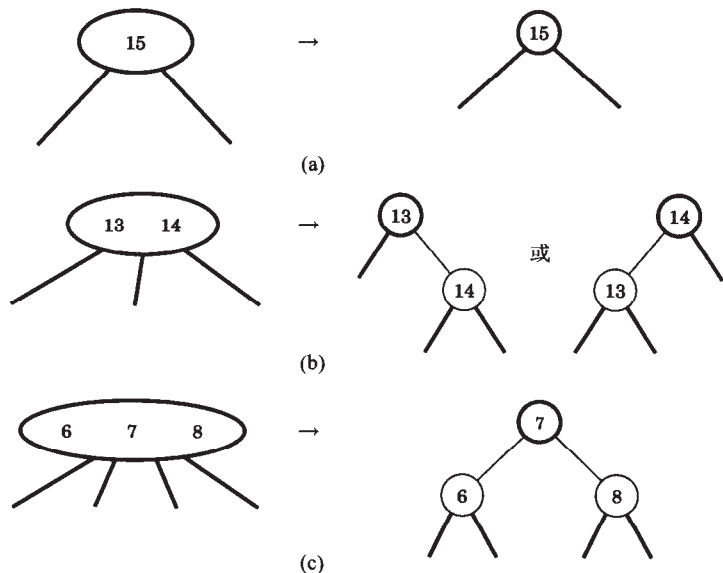
图3-19 与图3-13中的(2, 4)树关联的红黑树: 这棵红黑树的每个外部结点都有三个黑色祖先结点; 因此, 黑色深度为3。回忆一下我们利用粗线表示黑色结点

作为本章中的约定, 假设数据项存储在红黑树的内部结点中, 并且外部结点是空的占位符。同时, 描述算法时假设外部结点是真实的, 但注意到传递时, 要进行稍微复杂一点的查找和更新操作, 外部结点能够为null, 或指向NULL_NODE对象的引用。

170

注意到红黑树和(2, 4)树之间的有趣联系, 可以更直观地理解红黑树的定义。如图3-20中的说明。即给定一棵红黑树, 通过将每个红色结点 v 合并进它的父结点中, 并将 v 中的数据项存放到父结点中, 可以构造相应的(2, 4)树。相反, 通过对每个结点着黑色, 并对每个内部结点 v 进行如下简单的变换:

- 如果 v 是一个2-结点, 那么保持 v 的黑色子结点不变。
- 如果 v 是一个3-结点, 那么创建一个新的红色结点 w , 并将 v 的前两个黑色子结点给予 w , 使 w 和 v 的第三个子结点成为 v 的两个子结点。
- 如果 v 是一个4-结点, 那么创建两个新的红色结点 w 和 z , 并将 v 的前两个黑色子结点给予 w , 将 v 的后两个黑色子结点给予 z , 并使 w 和 z 成为 v 的两个子结点。



171

图3-20 (2, 4)树和红黑树之间的对应关系: (a)2-结点; (b)3-结点; (c)4-结点

(2, 4)树和红黑树之间的对应关系提供了将在讨论中使用的重要直觉。事实上, 红黑树的更新算法极其复杂, 而不具这种直觉。红黑树还具有如下性质。

定理3.5 存储 n 个数据项的红黑树的高度为 $O(\log n)$ 。

证明 设 T 是存储 n 个数据项的红黑树, h 是 T 的高度。通过建立如下事实, 证明这个定理:

$$\log(n+1) \leq h \leq 2\log(n+1)$$

设 d 是 T 的所有外部结点的公共黑色深度。设 T' 是关联 T 的(2, 4)树, h' 是 T' 的高度。可知 $h' = d$ 。因此, 由定理3.4可知, $d = h' \leq \log(n+1)$ 。由内部结点性质可知, $h \leq 2d$ 。因此, 可得 $h \leq 2\log(n+1)$ 。由定理2.8和 T 有 n 个内部结点这一事实可得另一个不等式: $\log(n+1) \leq h$ 。■

假设用二叉树的链表结构(2.3.4节)实现红黑树, 其中在每个结点上存储字典数据项和颜色标志。因此, 存储 n 个关键字所需空间为 $O(n)$ 。在一棵红黑树 T 中进行查找的算法与在一棵标准二叉查找树中的查找算法(3.1.2节)相同。因此, 红黑树中的查找时间为 $O(\log n)$ 。

在红黑树中进行更新操作类似于在二叉查找树中进行更新操作, 只不过在红黑树中要存储额外的颜色信息。

1. 红黑树中的插入

考虑将关键字为 k 的元素 x 插入一棵红黑树 T 中。记住 T 与其关联的(2, 4)树 T' 的对应关系, 以及 T' 的插入算法。插入算法初始时如同二叉查找树中的插入操作(3.1.4节)。即在 T 中查找 k , 直到达到 T 的一个外部结点, 用存储 (k, x) 及具有两个外部子结点的内部结点 z 代替这个结点, 如果 z 是树 T 的根, 则 z 着黑色, 否则 z 着红色。同时将 z 的子结点着黑色。这个行为对应于将数据项 (k, x) 插入(2, 4)树 T' 中一个带有外部结点的结点中。此外, 这个行为会保持 T 的根性质、外部性质和深度性质, 但它可能违反内部性质。实际上, 如果 z 不是 T 的根, 且 z 的父结点 v 是红色, 那么就有父结点和子结点(即 v 和 z)都是红色的结点。注意, 由根性质可知, v 不可能是 T 的根, 由内部性质(以前满足的)可知, v 的父结点 u 一定是黑色。因为 z 和它的父结点是红色, 但 z 的祖先结点 u 是黑色, 称这种对内部性质的违反为结点 z 处的双红色(double red)。

172

为了修正双红色问题, 考虑两种情况。

情况1: v 的兄弟结点 w 是黑色(如图3-21所示)。在这种情况下, 双红色表示这样一个事实, 即在红黑树中对应的(2, 4)树的4-结点上, 建立了一种畸形替换, 它把 u , v 和 z 的4个子结点作为它的子结点。畸形替换有一个红色结点(v), 它是另一个红色结点(z)的父结点, 而希望它有两个红色结点作为兄弟结点。为了修正这个问题, 对 T 进行三结点重构。通过操作 $\text{restructure}(z)$ 完成三结点重构, 该操作由以下几步组成(再次参见图3-21; 3.2节中也讨论了这个操作):

- 取结点 z 、它的父结点 v 、祖父结点 u , 并按照从左到右的顺序, 将它们暂时重新标记为 a 、 b 和 c , 以使得按照中序树遍历依次访问 a 、 b 和 c 。
- 用标为 b 的结点替换祖父结点 u , 使结点 a 和 c 成为 b 的子结点, 保持中序关系不变。

进行 $\text{restructure}(z)$ 操作之后, 对 b 着黑色, 对 a 和 c 着红色。因此, 重构消除了双红色问题。

173

情况2: v 的兄弟结点 w 是红色(如图3-22所示)。在这种情况下, 双红色表示对应的(2, 4)树 T 中的一个上溢。为了修正这个问题, 进行等价的分裂操作。也就是重新进行着色: 对 v 和 w 着黑色, 对它们的父结点 u 着红色(除非 u 是根, 在这种情况下, 对其着黑色)。可能在重新着色之后, 双红色问题再一次出现, 但出现在 T 的更高一层, 因为 u 可能有一个红色的父结点。如果双红色问题在 u 处再次出现, 那么在 u 处再次考虑这两种情况。因此, 重新着色要么消除了结点 z 处的双红色问题, 要么将它传播到 z 的祖父结点 u 处。继续向上进行重新着色的过程, 直到最终解决了双红色问题。

问题（进行最后的重新着色，或者进行三结点重构）。因此，由插入引起的重新着色次数不会超过树 T 高度的一半，即由定理3.5可知，重新着色次数不超过 $\log(n+1)$ 。

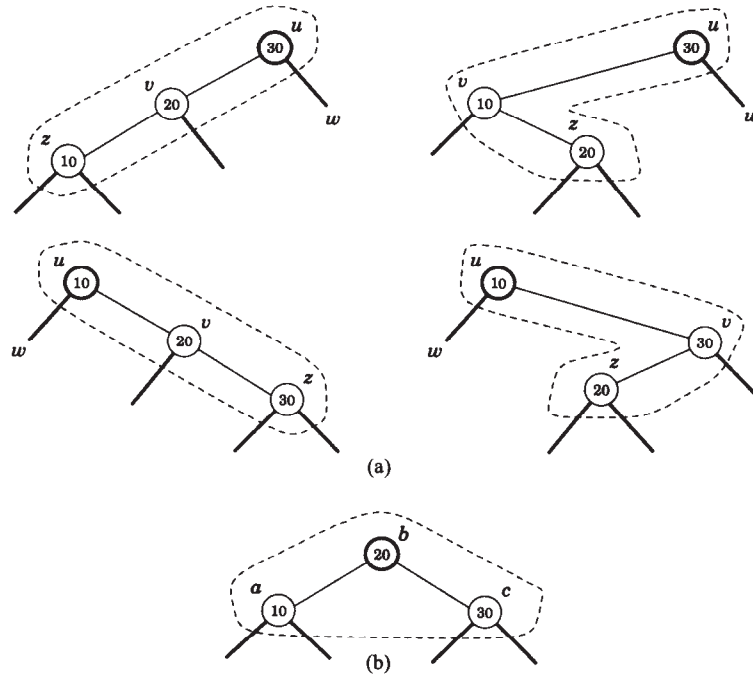


图3-21 重构红黑树，修正双红色问题：(a)重构之前， u ， v 和 z 的4种配置；(b)重构之后

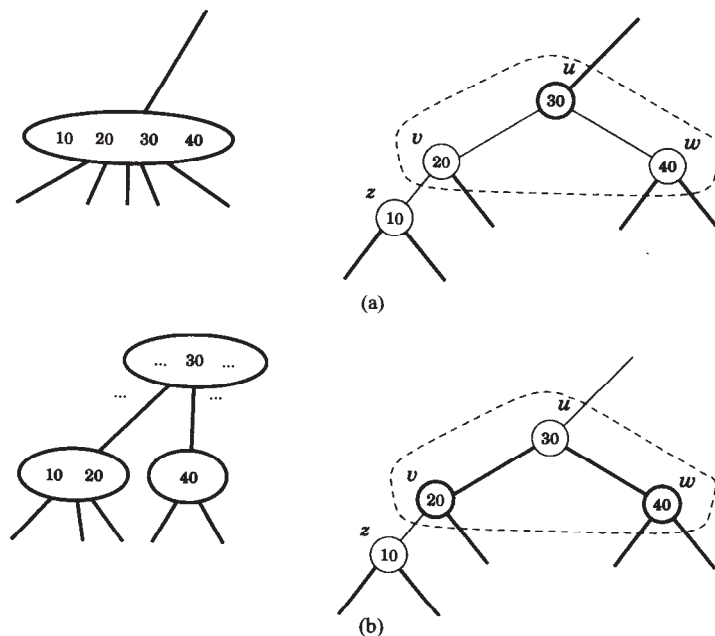


图3-22 重新着色修正双红色问题：(a)重新着色之前以及分裂前关联的(2, 4)树中对应的5-结点；(b)重新着色之后（以及分裂后关联的(2, 4)树中对应的结点）

图3-23和图3-24显示了红黑树中的插入序列。

174

插入情况蕴涵红黑树的有趣性质。也就是说，因为情况1的行为用一次三结点重构消除双红色问题，情况2的行为不进行重构操作，因此在红黑树的插入中至多需要一次重构。由以上分析及一次重构或重新着色所需时间为 $O(1)$ ，可得如下定理：

定理3.6 在存储 n 个数据项的一棵红黑树中，关键字-元素对数据项的插入时间为 $O(\log n)$ ，至多需要 $O(\log n)$ 次重新着色以及一次三结点重构（restructure操作）。

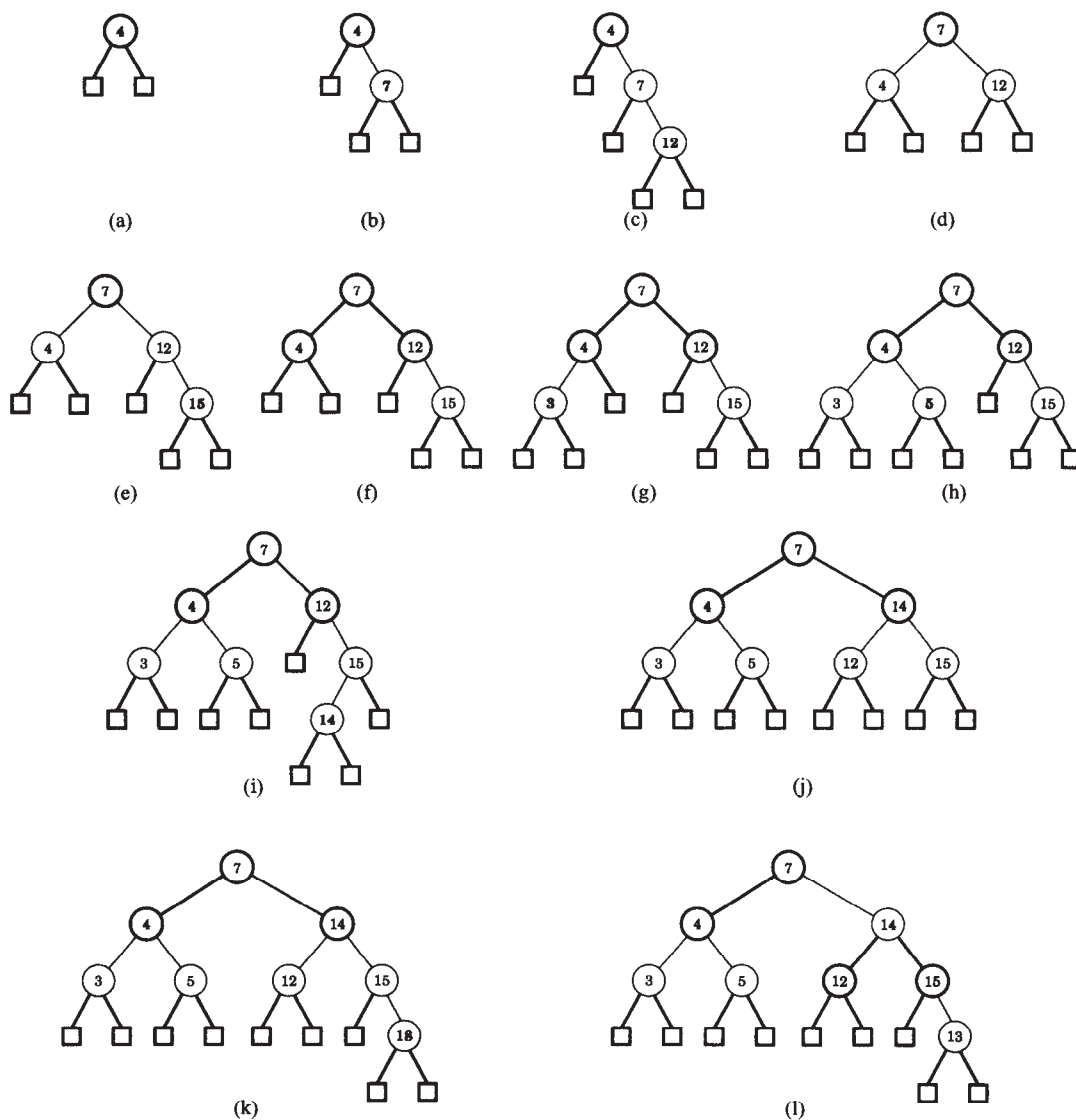
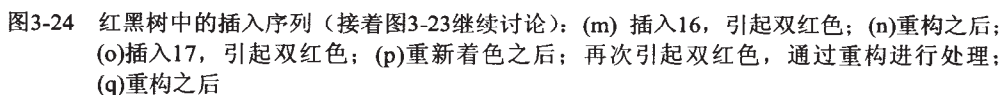


图3-23 红黑树中的插入序列：(a)初始树；(b)插入7；(c)插入12，引起双红色；(d)重构之后；(e)插入15，引起双红色；(f)重新着色之后（根依然为黑色）；(g)插入3；(h)插入5；(i)插入14，引起双红色；(j)重构之后；(k)插入18，引起双红色；(l)重新着色之后（图3-24继续讨论）

175



假定要从一棵红黑树 T 中删除关键字为 k 的数据项。初始时删除这样一个数据项就像在一棵二叉查找树中进行操作(3.1.5节)。首先查找存储这个数据项的结点 u 。如果结点 u 没有外部子结点,在对 T 的中序遍历中,找到紧跟在 u 后面的内部结点 v ,将 v 处的数据项移到 u ,并删除 v 中的数据项。因此,可能只考虑删除有一个外部子结点 w 的结点 v 中存储的关键字为 k 的数据项。正像在插入中所做的那样,记住红黑树 T 与其关联的(2,4)树 T' 的对应关系(以及 T' 的删除算法)。

为了从 T 的有一个外部子结点 w 的结点 v 中删除关键字为 k 的数据项, 操作方式如下。设 r 是 w 的兄弟结点, x 是 v 的父结点。删除结点 v 和 w , 并使 r 成为 x 的子结点。如果 v 是红色结点(由此 r 是黑色结点)或 r 是红色结点(由此 v 是黑色结点), 对 r 着黑色即可。如果 r 是黑色, v 是黑色, 那么为了保持深度性质, 给予 r 一个虚构的双黑色(double-black)。现在发生颜色违反, 称为双黑色问

题。 T 中的双黑色表示对应 $(2, 4)$ 树 T' 中的一个下溢。回忆可知, x 是双黑色结点 r 的父结点。为了修正结点 r 处的双黑色问题, 考虑三种情况。

情况1: r 的兄弟结点 y 是黑色, 且 y 有一个红色子结点 z 。如图3-25所示。这种情况对应于 $(2, 4)$ 树 T' 中的转移操作。利用操作 $\text{restructure}(z)$, 执行一次三结点重构。回忆可知, $\text{restructure}(z)$ 操作取结点 z 、其父结点 y 和祖父结点 x , 暂时将它们按照从左到右的次序标记为 a 、 b 和 c , 用标为 b 的结点代替 x , 使它成为另外两个结点的父结点。见3.2节对 restructure 的描述。对 a 和 c 着黑色, 使 b 的颜色为 x 以前的颜色, 并对 r 着黑色。这个三结点重构消除了双黑色问题。因此, 这种情况下的删除操作至多执行一次重构。

177

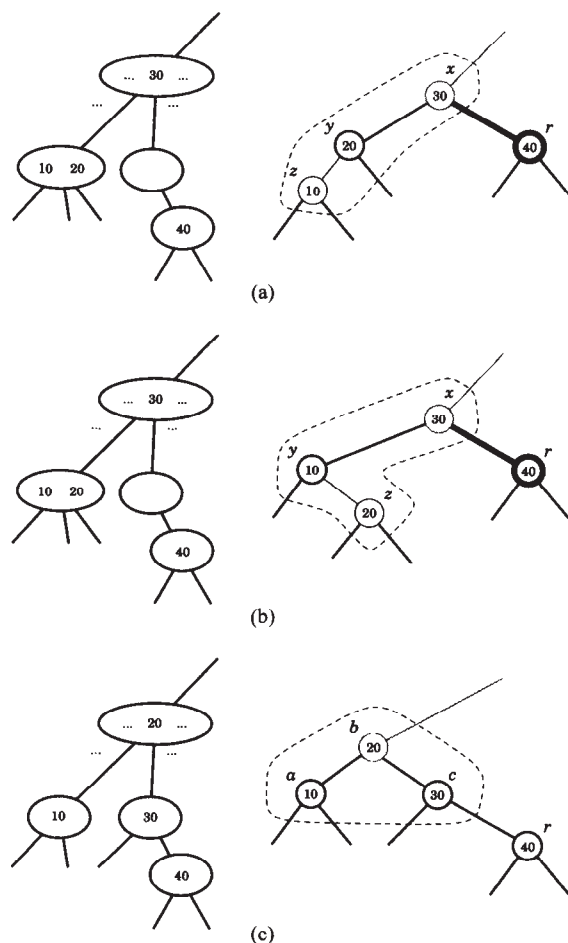


图3-25 重构红黑树, 修正双黑色问题: (a)和(b)重构之前的配置, 其中 r 是一个右子结点, 以及转移之前对应 $(2, 4)$ 树中的关联结点(r 是左子结点时, 可能存在另外两种对称配置); (c)重构后的配置, 以及转移后对应 $(2, 4)$ 树中的关联结点。(a)和(b)中的结点 x 以及(c)中的结点 b 可能为红色或者黑色

178

情况2: r 的兄弟结点 y 是黑色, 且 y 的两个子结点都是黑色。如图3-26和图3-27所示。解决这种情况对应于对应的 $(2, 4)$ 树 T' 中的合并操作。重新进行着色: 使 r 着黑色、 y 着红色, 且如果 x 为红色, 则对其着黑色(图3-26); 否则着双黑色(图3-27)。因此, 重新着色之后, 双黑色问题可能在 r 的父结点 x 处再次出现(图3-27)。也就是说, 这个重新着色要么消除双黑色问题, 要么将它

传播到当前结点的父结点中。然后，继续在父结点处考虑这三种情况。由于情况1执行一个三结点重构操作，并停止（正如将要看到的，情况3是类似的），因此，由删除引起的重新着色次数不超过 $\log(n+1)$ 。

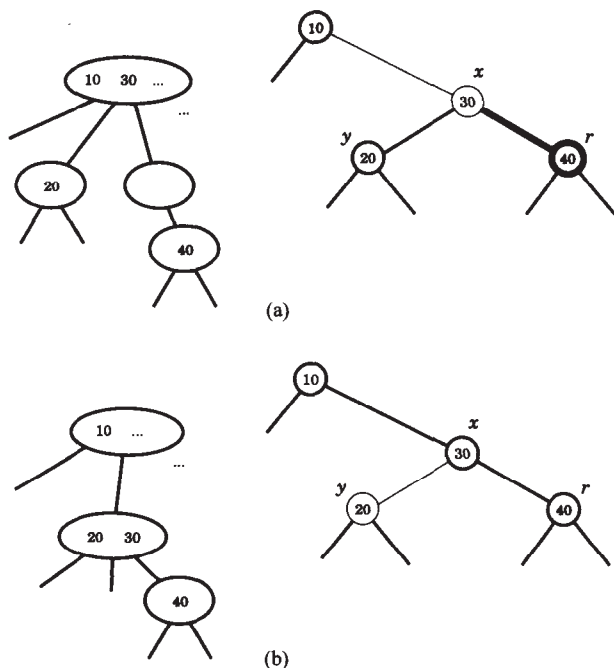


图3-26 重新着色红黑树，修正双黑色问题：(a)重新着色之前，以及合并之前关联的(2,4)树中的对应结点（还可能还有其他类似配置）；(b)重新着色之后，以及合并之后关联的(2,4)树中的对应结点

179

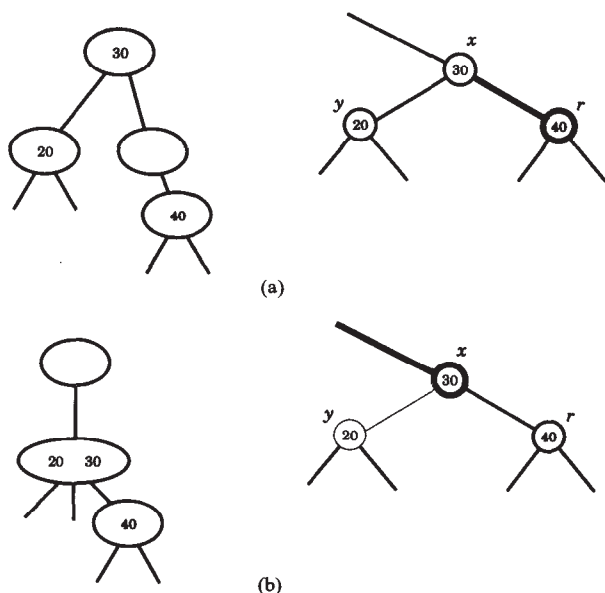


图3-27 传播双黑色问题的红黑树的重新着色：(a)重新着色之前的配置，以及合并之前关联的(2,4)树中的对应结点（还可能还有其他类似配置）；(b)重新着色之后的配置，以及合并之后关联的(2,4)树中的对应结点

180

情况3: r 的兄弟结点 y 是红色。如图3-28所示。在这种情况下,进行如下调整(adjustment)操作。如果 y 是 x 的右子结点,则令 z 是 y 的右子结点;否则,令 z 是 y 的左子结点。执行三结点重构操作restructure(z),使 y 成为 x 的父结点。对 y 着黑色,对 x 着红色。一次调整对应于选择(2, 4)树 T' 中3-结点的另一种表示。调整操作之后, r 的兄弟结点为黑色,则适用于情况1或情况2, x 和 y 具有不同的含义。注意如果应用情况2,双黑色问题不会再次出现。因此,为了完成情况3,多应用一次上述情况1或者情况2即可。于是,在一次删除操作中至多执行一次调整操作。

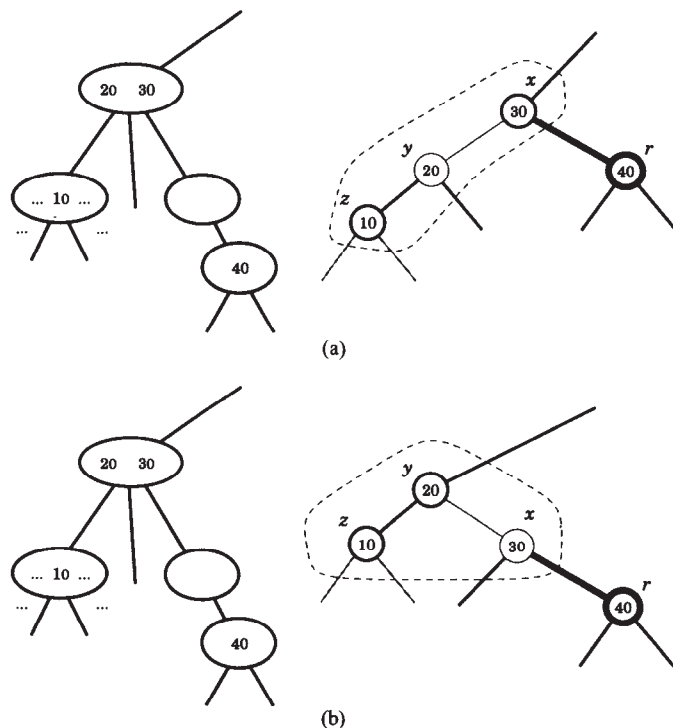


图3-28 出现双黑色问题时,对红黑树所进行的调整:(a)调整之前的配置,以及关联的(2, 4)树中的对应结点(可能还有对称配置);(b)调整之后的配置,以及关联的(2, 4)树中相同的对应结点

181

由上述算法描述可知,删除之后所需的树更新操作涉及树 T 中向上的操作过程,但每个结点上至多恒定的工作量(在一次重构、重新着色或调整中)。在这个向上的操作过程中,在 T 中任何结点上所做的改变所需时间均为 $O(1)$ 。因为它只影响常数个结点。此外,由于重构情况会终止树中向上的传播,可得如下定理。

定理3.7 从一棵具有 n 个数据项的红黑树中删除一个数据项的算法所需时间为 $O(\log n)$,并且会执行 $O(\log n)$ 次重新着色,以及至多一次调整加上另外一次三结点重构。因此,至多进行两次restructure操作。

在图3-29和图3-30中,显示了在一棵红黑树上进行删除操作的过程。图3-29c和图3-29d说明情况1的重构。在图3-29和图3-30的多个位置说明情况2的重新着色过程。最后,在图3-30i和图3-30j中显示了情况3的调整的一个例子。

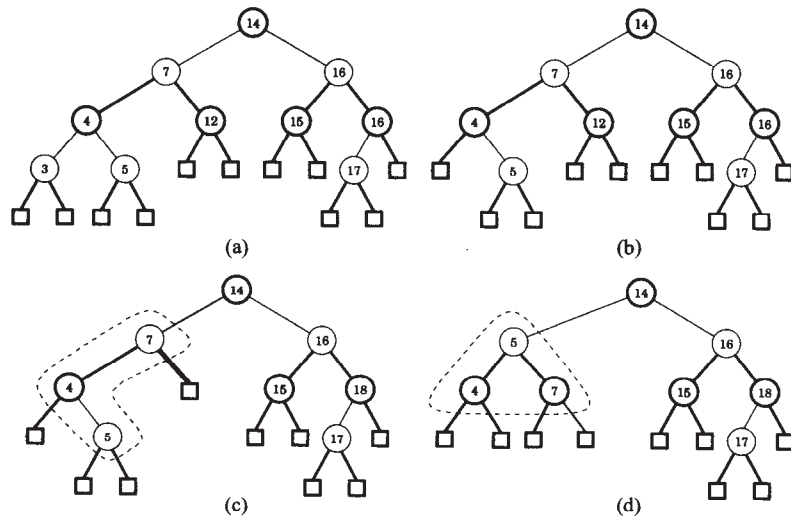


图3-29 红黑树中的删除序列: (a)初始树; (b)删除3; (c)删除12, 引起双黑色 (利用重构解决); (d)重构之后

182

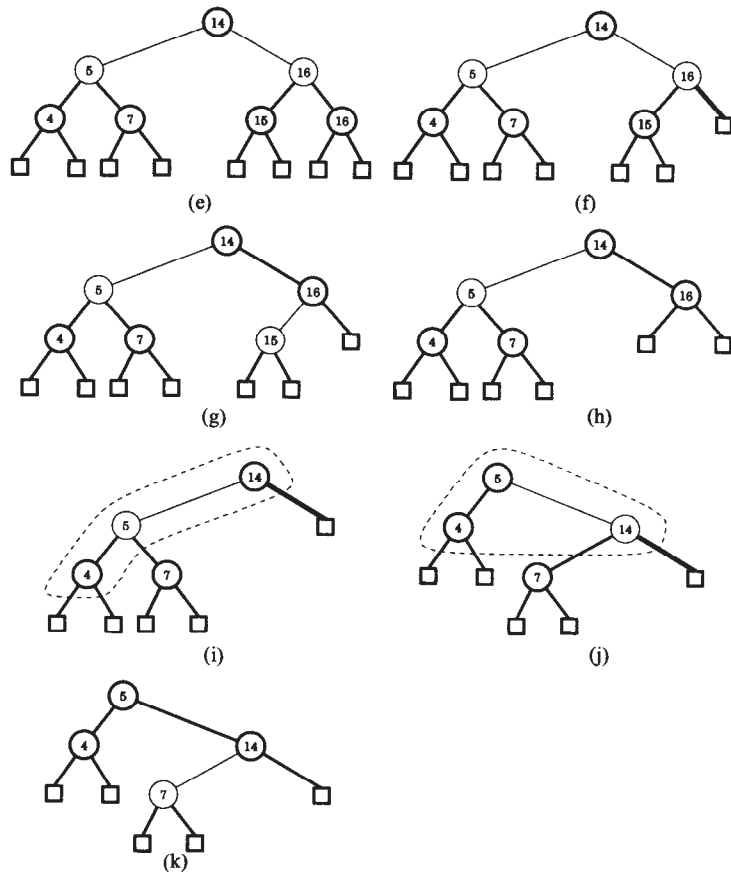


图3-30 红黑树中的删除序列 (续): (e)删除17; (f)删除18, 引起双黑色 (利用重新着色解决); (g)重新着色之后; (h)删除15; (i)删除16, 引起双黑色 (利用调整解决); (j)调整之后, 利用重新着色解决双黑色; (k)重新着色之后

183

3. 红黑树的性能

表3-4概括了红黑树实现的字典中主要操作的运行时间。图3-31说明了这些界限的正确性。

表3-4 红黑树实现的 n 个元素的字典的性能，其中 s 表示findAllElements和removeAllElements返回的迭代器大小

操 作	时 间
size、isEmpty	$O(1)$
findElement、insertItem、removeElement	$O(\log n)$
findAllElements、removeAllElements	$O(\log n + s)$

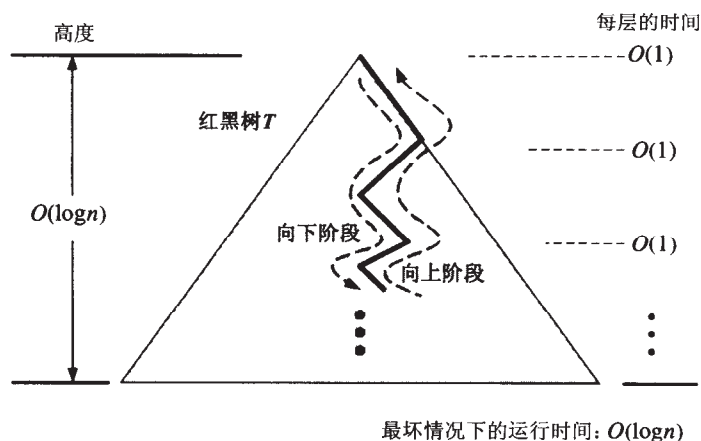


图3-31 红黑树中查找和更新时间说明。时间性能是每层 $O(1)$ ，分成与查找对应的向下阶段，与重新着色及执行局部三结点重构（旋转）的向上阶段

因此，在字典中进行查找和更新，红黑树最坏情况下将获得对数运行时间。红黑树的数据结构比其对应的(2, 4)树的数据结构稍微复杂一些。即便如此，红黑树在概念上有优势，因为更新之后，要在一棵红黑树中恢复平衡，只需进行常量次数的三结点重构操作。

184

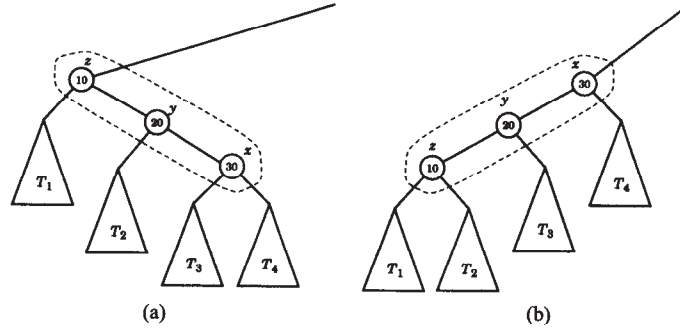
3.4 伸展树

本章讨论的最后一个平衡查找树数据结构是伸展树（splay tree）。这个结构在概念上完全不同于以前讨论的平衡查找树（AVL、红黑树和(2, 4)树），因为伸展树并不利用任何明确的规则，来保证它的平衡。而是在每次访问之后，利用一种称为伸展（splaying）的移动到根的操作，来保持查找树在平摊意义上的平衡。在插入、删除，甚至查找过程中，在到达的最底部的结点 x 上执行伸展操作。令人惊异的事情是，伸展可以保证插入、删除和查找的平摊运行时间是对数时间。伸展树的结构只是一棵二叉查找树 T 。事实上，在这种树的结点上，并没有关联额外的高度、平衡和颜色标签。

3.4.1 伸展

给定二叉查找树 T 的一个内部结点 x ，通过一系列重构过程，将结点 x 移到 T 的根处，伸展 x 。进行的特殊重构非常重要。因为只进行任意系列的重构，还不足以将 x 移到 T 的根。将 x 向上移动的特定操作取决于 x 、其父结点 y 及其祖父结点 z （若存在）的相对位置。考虑三种情况。

zig-zig: 结点 x 和它的父结点 y 都是左、右子结点。如图3-32所示。用 x 代替 z , 使 y 成为 x 的子结点, 并使 z 成为 y 的子结点, 同时维持 T 中结点的中序关系。



185

图3-32 zig-zig: (a)之前; (b)之后。存在另一种对称配置, 其中 x 和 y 都是左子结点

zig-zag: x 和 y 其中之一是左子结点, 另一个是右子结点。如图3-33所示。在这种情况下, 用 x 代替 z , 并使 y 和 z 作为 x 的子结点, 同时维持 T 中结点的中序关系。

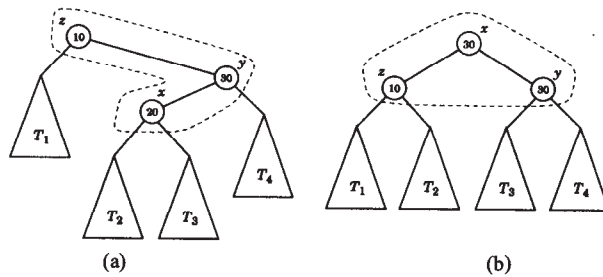


图3-33 zig-zag: (a)之前; (b)之后。当 x 是右子结点, y 是左子结点时, 存在另一种对称配置

zig: x 没有祖父结点 (或者出于某些原因, 不考虑 x 的祖父结点)。如图3-34所示。在这种情况下, 沿着结点 y 旋转 x , 使 x 的子结点是结点 y 和 x 以前的子结点中的一个结点 w , 以便维持 T 中结点的相对中序关系。

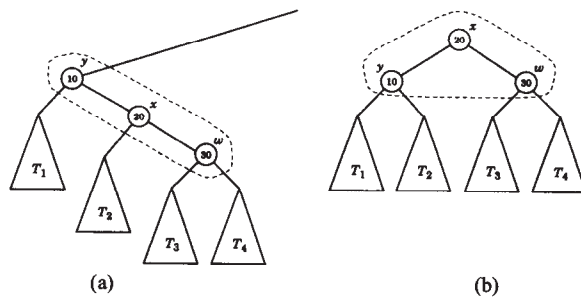


图3-34 zig: (a)之前; (b)之后。当 x 和 w 都是左子结点时, 存在另一种对称配置

当 x 有一个祖父结点时, 进行zig-zig操作或者zig-zag操作。当 x 有父结点但没有祖父结点时, 进行zig操作。伸展过程在结点 x 处不断进行重构, 直到 x 成为 T 的根。注意不存在像旋转那样使得 x 成为根的简单过程。伸展一个结点的例子如图3-35和图3-36所示。

一次zig-zig或zig-zag之后, x 的深度减2, 一次zig之后, x 的深度减1。因此, 如果 x 的深度为 d

并且 d 为基数，伸展 x 的过程由 $\lfloor d/2 \rfloor$ 个zig-zig和（或）zig-zag加上一个最终的zig组成。因为单个zig-zig、zig-zag或zig影响常数个结点，每个操作所需时间为 $O(1)$ 。因此，在一棵二叉查找树 T 中伸展一个结点 x 所需时间为 $O(d)$ ，其中 d 为 T 中结点 x 的深度。换句话说，在一个结点 x 上进行伸展所需的时间与从 T 的根进行自顶向下查找以到达该结点所需的时间相同。

186

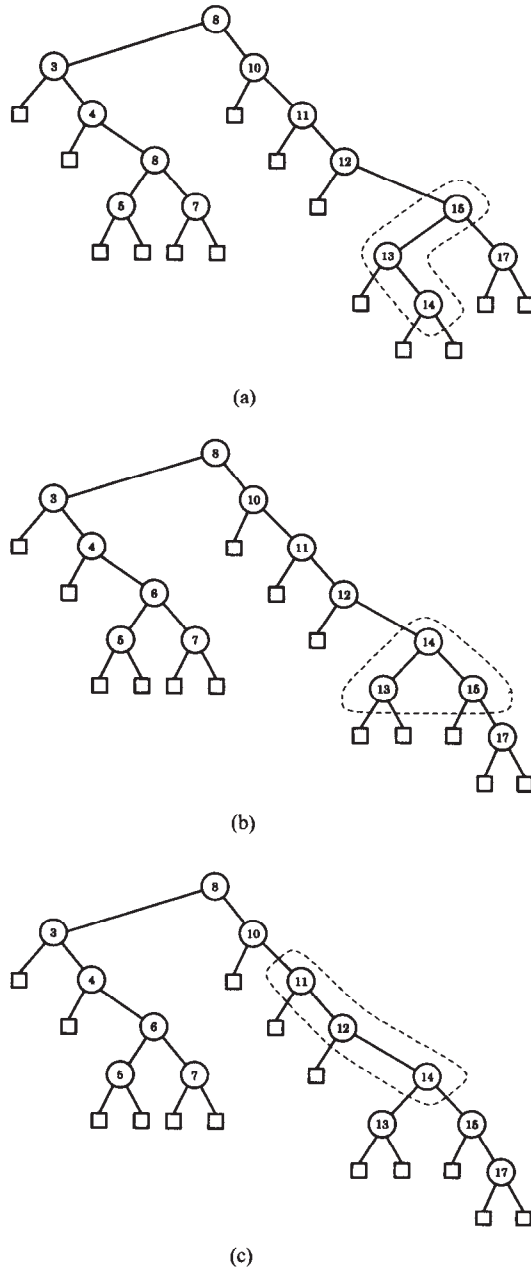


图3-35 伸展一个结点的示例：(a)用zig-zag开始伸展存储14的结点；(b)zig-zag之后；(c)下一步是zig-zig

187

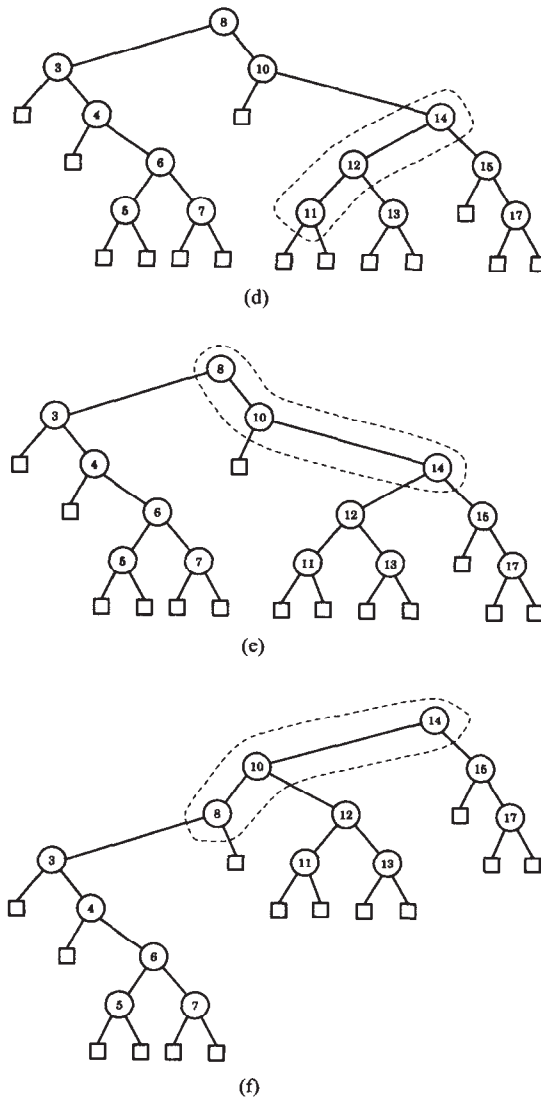


图3-36 伸展一个结点的示例（接着图3-35继续）：(d) zig-zig之后；(e)下一步又是zig-zig；(f)zig-zig之后

188

何时伸展

规则表明进行伸展时，执行如下操作：

- 查找关键字 k 时，如果在结点 x 处找到 k ，则伸展 x ，否则在查找不成功地终止的外部结点处，伸展该外部结点的父结点。例如，在图3-35和图3-36中，在成功查找到关键字14之后伸展，或者在不成功地查找到关键字14.5之后伸展。
- 插入关键字 k 时，伸展插入 k 时新建立的内部结点。例如，图3-35和图3-36中进行伸展，当插入新关键字14时。图3-37显示伸展树中的插入序列。
- 删除关键字 k 时，伸展要被删除的结点 w 的父结点，即 w 或者是存储 k 的结点，或者是其后代之一（回忆3.1.2节给出的二叉查找树的删除算法）。图3-38显示了删除之后伸展的例子。

189

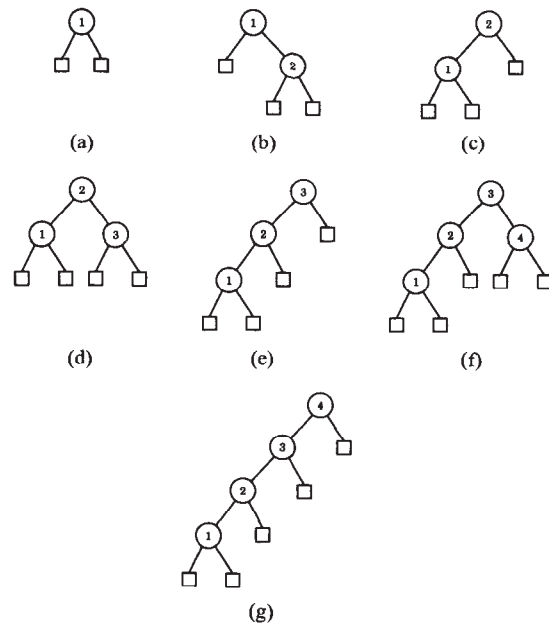


图3-37 伸展树中的插入序列: (a)初始树; (b)插入2后; (c)伸展后; (d)插入3后; (e)伸展后; (f)插入4后; (g)伸展后

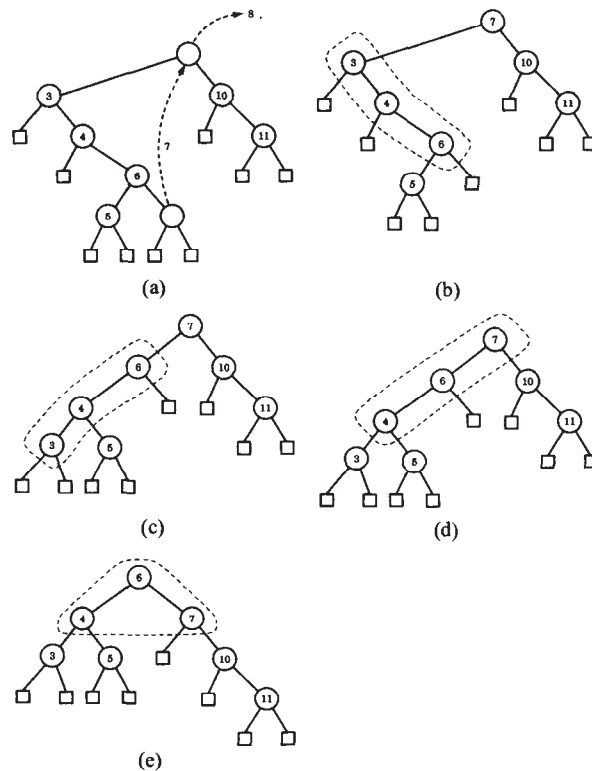


图3-38 伸展树中的删除过程: (a)将最右边的内部结点 v 的关键字移到结点 r , 从 r 中删除8, 在 r 的左子树中, 删除 v , 伸展 v 的父结点 u ; (b)用zig-zig开始伸展 u ; (c)zig-zig之后; (d)下一步是zig; (e)zig之后

190

在最坏情况下，在一棵高度为 h 的伸展树中进行查找、插入或者删除的总运行时间为 $O(h)$ ，因为被伸展的结点可能是树中最深的结点。此外， h 可能为 $\Omega(n)$ 。如图3-37所示。因此，从最坏情况下的角度看，伸展树并不是一种有吸引力的数据结构。

3.4.2 伸展过程的平摊分析

尽管伸展树最坏情况下的性能不佳，但是在平摊意义上，伸展树还是进行得很好。也就是说，在查找、插入和删除操作的混合序列中，每个操作平均花费对数时间。我们注意到，进行一次查找、插入或者删除操作的时间与关联的伸展时间成正比；因此，在以下的分析中，只考虑伸展时间。

设 T 是一棵有 n 个关键字的伸展树， v 是 T 中的一个结点。定义 v 的大小（size） $n(v)$ 是以 v 为根的子树中的结点数。注意一个内部结点的大小等于它的两个子结点的大小之和再加1。定义结点 v 的位序（rank） $r(v)$ 为以2为底的 v 的大小的对数，即 $r(v) = \log n(v)$ 。显然， T 的根的最大大小为 $2n+1$ ，最大位序为 $\log(2n+1)$ ，而每个外部结点的大小为1，位序为0。

利用计算机元支付伸展 T 中的一个结点 x 所进行的工作，假设一个计算机元支付一次zig，而两个计算机元则支付一次zig-zig或zig-zag。因此，伸展一个深度为 d 的结点的成本为 d 个计算机元。在 T 的每个内部结点上，保持一个虚设的账户，存储计算机元。注意这个账户仅出于平摊分析的目的而存在，在实现伸展树 T 的数据结构中并不需要包括它。当进行伸展时，支付一定数量的计算机元（后面将确定精确值）。分三种情况讨论：

- 如果付款等于伸展工作，那么用它的所有付款支付伸展。
- 如果付款大于伸展工作，将多余部分储蓄在多个结点的账户中。
- 如果付款小于伸展工作，从多个结点的账户中取款，弥补差额。

将证明每个操作支付的 $O(\log n)$ 计算机元足以保持系统工作，即确保每个结点保持非负的账目余额。可以利用一种模式，在这种模式中可以在结点的账户之间进行转账，确保提取足够的计算机元，用以支付所需的伸展工作。同时，维持以下不变式：

在伸展之前和之后， T 的每个结点 v 都有 $r(v)$ 个计算机元

注意，不变式并不要求向空树赠与计算机元。设 $r(T)$ 是 T 中所有结点的位序之和。为了保持伸展之后的不变式，必须使付款等于伸展工作加上 $r(T)$ 中总的变化。称伸展中单个zig、zig-zig或zig-zag操作为一个伸展子步（substep）。同时，用 $r(v)$ 和 $r'(v)$ 分别表示伸展子步之前和之后， T 中结点 v 的位序。以下引理给出由单个伸展子步引起的 $r(T)$ 变化的上界。

191

引理3.1 设 δ 是对伸展树 T 中的结点 x 执行单个伸展子步（zig、zig-zig或zig-zag）而引起的 $r(T)$ 的变化，则有

- 如果子步是zig-zig或zig-zag，则 $\delta \leq 3(r'(x) - r(x)) - 2$ 。
- 如果子步是zig，则 $\delta \leq 3(r'(x) - r(x))$ 。

证明 利用以下数学事实（见附录A）：

如果 $a > 0$ ， $b > 0$ ， $c > a + b$ ，那么

$$\log a + \log b \leq 2 \log c - 2 \quad (3.6)$$

考虑每种类型的伸展子步引起 $r(T)$ 的变化。

zig-zig:（回忆图3-32）因为每个结点的大小是其两个子结点的大小之和再加1。注意在一次zig-zig操作中，只有 x 、 y 和 z 的位序发生变化，其中 y 是 x 的父结点，而 z 是 y 的父结点。同时， $r'(x) =$

$r(z)$, $r'(y) \leq r'(x)$, 且 $r(y) \geq r(x)$ 。因此,

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\leq r'(y) + r'(z) - r(x) - r(y) \\ &\leq r'(x) + r'(z) - 2r(x)\end{aligned}\quad (3.7)$$

观察可得, $n(x) + n'(z) \leq n'(x)$ 。因此, 由3.6可知, $r(x) + r'(z) \leq 2r'(x) - 2$ 。

也就是说,

$$r'(z) \leq 2r'(x) - r(x) - 2$$

由此不等式和公式(3.7)可知

$$\begin{aligned}\delta &\leq r'(x) + (2r'(x) - r(x) - 2) - 2r(x) \\ &\leq 3(r'(x) - r(x)) - 2\end{aligned}$$

zig-zag: (回忆图3-33) 再次依据大小和位序的定义, 只有 x 、 y 和 z 的位序发生变化, 其中 y 表示 x 的父结点, z 表示 y 的父结点。同时, $r'(x) = r(z)$ 和 $r(x) \leq r(y)$ 。因此,

$$\begin{aligned}\delta &= r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) \\ &\leq r'(y) + r'(z) - r(x) - r(y) \\ &\leq r'(y) + r'(z) - 2r(x)\end{aligned}\quad (3.8)$$

观察可得, $n'(y) + n'(z) \leq n'(x)$ 。因此, 由3.6可知, $r'(y) + r'(z) \leq 2r'(x) - 2$ 。

由此不等式和公式(3.8)可知

$$\begin{aligned}\delta &\leq 2r'(x) - 2 - 2r(x) \\ &\leq 3(r'(x) - r(x)) - 2\end{aligned}$$

zig: (回忆图3-34) 在这种情况下, 只有 x 和 y 的位序发生变化, 其中 y 表示 x 的父结点。同时, $r'(y) \leq r(y)$ 和 $r'(x) \geq r(x)$ 。因此,

$$\begin{aligned}\delta &= r'(y) + r'(x) - r(y) - r(x) \\ &\leq r'(x) - r(x) \\ &\leq 3(r'(x) - r(x))\end{aligned}$$

■ 192

定理3.8 设 T 是根为 t 的伸展树, Δ 是伸展深度为 d 的结点 x 引起 $r(T)$ 的总变化, 则有

$$\Delta \leq 3(r(t) - r(x)) - d + 2$$

证明 伸展结点 x 由 $p = \lceil d/2 \rceil$ 个伸展子步组成, 每个子步都是一个zig-zig或zig-zag (可能除了最后一个子步之外, 如果 d 为奇数, 则它是一个zig)。设 $r_0(x) = r(x)$ 为 x 的初始位序, 对于 $i = 1, \dots, p$, 设 $r_i(x)$ 是第 i 个子步后 x 的位序, δ_i 是第 i 个子步引起的 $r(T)$ 的变化。由引理3.1可知, 伸展结点 x 引起 $r(T)$ 的总变化如下:

$$\begin{aligned}\Delta &= \sum_{i=1}^p \delta_i \\ &\leq \sum_{i=1}^p (3(r_i(x) - r_{i-1}(x)) - 2) + 2 \\ &= 3(r_p(x) - r_0(x)) - 2p + 2 \\ &\leq 3(r(t) - r(x)) - d + 2\end{aligned}$$

■

由定理3.8可知, 如果为结点 x 的伸展支付 $3(r(t) - r(x)) - d + 2$ 个计算机元, 则有足够计算机元

维持不变式。在 T 的每个结点 v 上保持 $r(v)$ 个计算机元, 支付整个伸展工作, 这要花费 d 元。因为根 t 的大小为 $2n+1$, 位序为 $r(t) = \log(2n+1)$ 。此外, 有 $r(x) < r(t)$ 。因此, 伸展需要支付 $O(\log n)$ 个计算机元。为了完成分析, 必须计算当插入或删除结点时, 维持不变式的成本。

当向有 n 个关键字的伸展树中插入一个新结点 v 时, v 的所有祖先结点的位序都会增加。也就是说, 设 v_0, v_1, \dots, v_d 是 v 的祖先结点, 其中 $v_0 = v$, v_i 是 v_{i-1} 的父结点, v_d 是根。对于 $i = 1, \dots, d$, 设 $n'(v_i)$ 和 $n(v_i)$ 分别是插入前后 v_i 的大小, $r'(v_i)$ 和 $r(v_i)$ 分别是插入前后 v_i 的位序。则有

$$n'(v_i) = n(v_i) + 1$$

同时, 因为 $n(v_i) + 1 \leq n(v_{i+1})$, 对于 $i = 0, 1, \dots, d-1$, 对于此范围中的每个 i , 可得:

$$r'(v_i) = \log(n'(v_i)) = \log(n(v_i) + 1) \leq \log(n(v_{i+1})) = r(v_{i+1})$$

因此, 插入引起的 $r(T)$ 的总变化为

$$\begin{aligned} \sum_{i=1}^d (r'(v_i) - r(v_i)) &\leq r'(v_d) + \sum_{i=1}^{d-1} (r(v_{i+1}) - r(v_i)) \\ &= r'(v_d) - r(v_0) \\ &\leq \log(2n+1) \end{aligned}$$

193

因此, 当插入新结点时, 支付 $O(\log n)$ 个计算机元足以维持不变式。

当在有 n 个关键字的伸展树中删除结点 v 时, v 的所有祖先的位序都会减少。因此, 由删除引起 $r(T)$ 的总变化为负, 不需支付计算机元就可维持不变式。于是, 以下定理概括了平摊分析。

定理3.9 从没有一个关键字的空伸展树开始, 考虑伸展树上的 m 个操作序列, 这些操作都是查找、插入或删除。同时, 设 n_i 是第 i 个操作后, 树中的关键字数, n 是插入总次数。执行这个操作序列的总运行时间为

$$O\left(m + \sum_{i=1}^m \log n_i\right)$$

即为 $O(m \log n)$ 。

换句话说, 在伸展树中进行查找、插入或删除的平摊运行时间为 $O(\log n)$, 其中 n 是当时伸展树的大小。因此, 对于实现有序字典ADT的伸展树, 其平摊性能可达到对数时间。这个平均性能与AVL树、(2,4)树和红黑树最坏情况下的性能相当。但它只使用简单二叉树, 无需在其每个结点处存储任何额外的平衡信息。此外, 伸展树还有许多其他平衡查找树不具有的有趣性质, 以下定理给出了其中的一个性质 (该定理有时称为伸展树的“静态最优性”定理)。

定理3.10 从没有一个关键字的树 T 开始, 考虑伸展树上的 m 个操作序列, 这些操作都是查找、插入或删除。此外, 设 $f(i)$ 表示伸展树中数据项 i 被访问的次数, 即它的频率 (frequency), 设 n 是数据项总数。假设每个数据项至少被访问一次, 那么执行操作序列的总运行时间为

$$O\left(m + \sum_{i=1}^n f(i) \log(m/f(i))\right)$$

这个定理的证明留作习题。该定理阐述了访问数据项 i 的平摊运行时间为 $O(\log(m/f(i)))$ 。例如, 如果一个操作序列访问某个数据项可达 $m/4$ 次之多, 当用伸展树实现字典时, 那么这些访问中的每一个的平摊运行时间为 $O(1)$ 。与之形成对比的是, 当用AVL树、(2,4)树或者红黑树实现字典时, 那么访问这个数据项的平摊运行时间为 $\Omega(\log n)$ 。因此, 伸展树的美好性质是它们能够“适合”

字典中数据项被访问的方式,以便得到对于频繁访问数据项的更快运行时间。

194

3.5 跳跃表

跳跃表(skip list)是一种有效实现有序字典ADT的数据结构。这种数据结构按照随机选择排列数据项,其上的查找和更新时间平均为对数时间。

1. 随机化数据结构和算法

这里使用的平均时间复杂度的概念并不依赖于输入中关键字的概率分布;而是依赖于使用的随机数生成器,它在实现插入时有助于确定新数据项的插入位置。也就是说,数据结构的结构及其上运行的一些算法依赖于随机事件的结果。在这种环境中,运行时间为插入数据项时所用的随机数的所有可能结果的平均值。

由于随机数广泛用于计算机游戏、密码学和计算机模拟中,因而可以生成随机数的方法已经构建在大多数现代计算机中。某些方法称为伪随机数生成器(pseudo-random number generator),它们从一个称为种子(seed)的初始数开始,明确产生类似于随机的数。还有一些方法利用硬件设备提取自然中的“真正”随机数。在任何情况下,假设计算机访问的数对于我们的分析是足够随机的。

数据结构和算法设计中利用随机化(randomization)的主要优点是,得到的结构和方法简单而有效。我们能够设计一个简单的随机化数据结构,称之为跳跃表,且查找的时间界限为对数时间。类似于二分搜索算法得到的结果。然而,对数界限是跳跃表的期望值,而在查找表中进行二分搜索则会出现最坏情况(worst-case)。另一方面,对于字典更新操作,跳跃表要比查找表快得多。

2. 跳跃表定义

字典 D 的跳跃表 S 由一系列表 $\{S_0, S_1, \dots, S_h\}$ 组成。每个表 S_i 存储 D 中数据项的一个子集,这些数据项按照非降关键字以及两个特殊关键字 $-\infty$ 和 $+\infty$ 排序,其中 $-\infty$ 小于插入 D 中可以插入的每个可能的关键字, $+\infty$ 则大于插入 D 中可以插入的每个可能的关键字。此外, S 中的表满足以下条件:

- 表 S_0 包含字典 D 的每个数据项(另加具有关键字 $-\infty$ 和 $+\infty$ 的特殊数据项)。
- 对于 $i = 1, \dots, h-1$,表 S_i 包含表 S_{i-1} 中随机生成的数据项的子集(除 $-\infty$ 和 $+\infty$ 之外)。
- 表 S_h 只含 $-\infty$ 和 $+\infty$ 。

195

跳跃表的一个例子如图3-39所示。通常将跳跃表 S 看作表 S_0 在底部,表 S_1, \dots, S_h 则在其上。此外,称 h 为跳跃表 S 的高度。

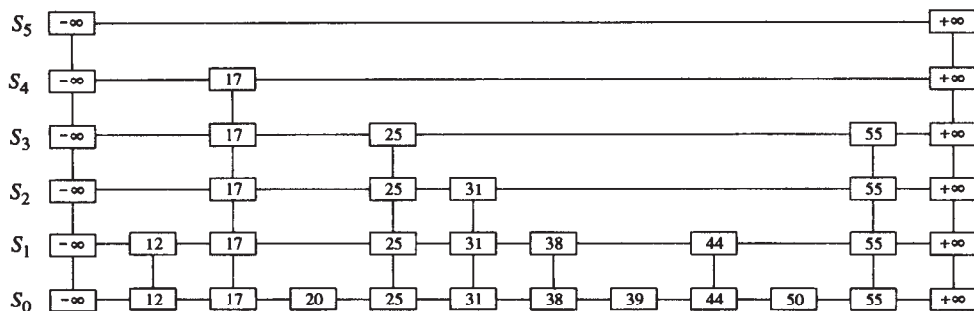


图3-39 跳跃表示例

直观上讲, 建立表 S_{i+1} 时, 或多或少会包含 S_i 中的数据项。正如将要在插入方法的细节中看到的那样, 从 S_i 中以概率 $1/2$ 随机选择每个数据项, 并将其加入 S_{i+1} 中。也就是说, 实质上, 对 S_i 中的数据项“投掷硬币”, 如果出现硬币的“正面(head)”, 则将该数据项放入 S_{i+1} 中。因此, 期望 S_1 中有大约 $n/2$ 个数据项, S_2 中有大约 $n/4$ 个数据项, 一般而言, S_i 中有大约 $n/2^i$ 个数据项。换句话说, 期望 S 的高度 h 约为 $\log n$ 。但是, 跳跃表并不要求从一个表到下一个表, 其数据项数要减少一半, 而是使用随机化。

利用表和树中使用的位置抽象, 将跳跃表看作二维位置集合, 该集合水平排列成层(level), 垂直排列成塔(tower)。每一层是一个表 S_i , 每个塔包含跨连续表存储相同数据项的位置。可以利用以下操作遍历跳跃表中的位置:

- **after(p)**: 返回同一层上 p 之后的位置。
- **before(p)**: 返回同一层上 p 之前的位置。
- **below(p)**: 返回同一塔中 p 之下的位置。
- **above(p)**: 返回同一塔中 p 之上的位置。

如果请求的位置不存在, 通常假设上述操作返回null位置。不需探究细节, 给定跳跃表位置 p , 可以容易地利用链表结构实现一个跳跃表, 满足上述每个遍历方法花费的时间 $O(1)$ 。这样的链表结构实际上是按塔排列的 h 个双向链表的集合, 而塔本身也是双向链表。

196

3.5.1 查找

跳跃表允许用于简单的字典查找算法。事实上, 跳跃表的所有查找方法基于一个优雅的SkipSearch过程, 如算法3-4所示。取一个关键字 k , 在跳跃表中找出具有小于或等于关键字 k 的最大关键字(可能为 $-\infty$)的数据项。

算法3-4 在跳跃表 S 中进行查找的算法

```

算法 SkipSearch( $k$ ):
  输入: 查找关键字 $k$ 
  输出:  $S$ 中的位置, 它的数据项具有小于或等于 $k$ 的最大关键字
  设 $p$ 在 $S$ 的最左上位置( $S$ 至少应该具有2层)
  while below( $p$ )  $\neq$  null do
     $p \leftarrow$  below( $p$ )      {下落}
    while key(after( $p$ ))  $\leq k$  do
      令 $p \leftarrow$  after( $p$ )    {前向扫描}
  return  $p$ .
```

仔细考察这个算法。首先, 在跳跃表 S 中将位置变量 p 设置在最左上位置, 开始SkipSearch方法。也就是说, p 被设置在 S_h 中关键字为 $-\infty$ 的特殊数据项所在的位置。然后执行如下步骤(如图3-40所示):

(1) 如果 $S.below(p)$ 为null, 那么查找终止——位于底部(at the bottom), 并且已经找到 S 中其关键字小于或等于查找关键字 k 的最大数据项位置。否则, 设置 $p \leftarrow S.below(p)$, 在当前塔中下移(drop down)一层。

(2) 从位置 p 开始, 前向移动 p , 直至到达当前层最右边的位置, 满足 $key(p) \leq k$ 。称之为前向扫描(scan forward)步骤。注意, 这样的位置总是存在, 因为每一层都包含特殊关键字 $-\infty$ 和 $+\infty$ 。事实上, 对这一层进行前向扫描之后, p 可能仍然保持在其开始的位置。无论如何, 接着重复上述步骤。

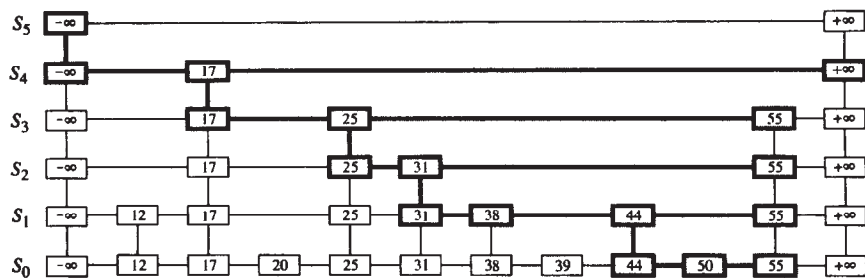


图3-40 跳跃表中的查找示例。查找关键字52（不成功）时访问的位置和遍历的链接用粗线表示

197

3.5.2 更新操作

给定SkipSearch方法，易于实现findElement(k)。只需简单地执行 $p \leftarrow \text{SkipSearch}(k)$ ，并测试是否 $\text{key}(p) = k$ 。正如所证明的那样，SkipSearch算法期望运行时间为 $O(\log n)$ 。不过，在讨论跳跃表的更新方法之后，再对其进行分析。

1. 插入

跳跃表的插入算法利用随机化，确定应该把多少个指向新数据项(k, e)的引用添加到跳跃表中。首先，执行SkipSearch(k)操作将新数据项(k, e)插入到跳跃表中。这给出具有小于或等于 k 的最大关键字的底层数据项的位置 p (p 可能是关键字为 $-\infty$ 的特殊数据项的位置)。然后将(k, e)插入底层表中并紧接在位置 p 的后面。在此层上插入新的数据项之后，“投掷”一枚硬币，即调用方法random()，它返回一个0~1之间的数。如果此数小于1/2，则认为出现“正面”；否则，认为出现“反面(tail)”。如果投掷出现反面，则在此停止。另一方面，如果投掷出现正面，则回溯到上一层(下一个更高层)，在该层的合适位置插入数据项(k, e)。再次投掷硬币，如果出现正面，则转到下一个更高层，并重复这个过程。因此，继续在表中插入新数据项(k, e)，直到最终投掷出现反面。把这个过程中所建立的对新数据项的所有引用链接在一起，建立(k, e)的一个塔。跳跃表 S 的插入算法的伪代码如算法3-5所示。图3-41对这个算法进行了说明。插入算法利用操作insertAfterAbove($p, q, (k, e)$)，它在位置 p 后(与 p 同层)、 q 之上插入存储数据项(k, e)的位置，并返回新数据项的位置 r (设置内部引用，使得after、before、above和below方法对于 p, q, r 能正确工作)。

算法3-5 跳跃表中的插入，假设random()返回一个0~1之间的随机数，且从不在顶层后插入

```

算法 SkipInsert( $k, e$ ):
  输入: 数据项( $k, e$ )
  输出: 无
   $p \leftarrow \text{SkipSearch}(k)$ 
   $q \leftarrow \text{insertAfterAbove}(p, \text{null}, (k, e))$  {在底层}
  while random() < 1/2 do
    while above( $p$ ) = null do
       $p \leftarrow \text{before}(p)$  {向后扫描}
     $p \leftarrow \text{above}(p)$  {跳向高级}
     $q \leftarrow \text{insertAfterAbove}(p, q, (k, e))$  {插入新数据项}

```

198

2. 删除

像查找和插入算法一样，跳跃表 S 上的删除算法也很简单。事实上，它甚至比插入算法还要简单，即为了执行removeElement(k)操作，首先执行对给定关键字 k 的查找。如果没找到关键字为

k 的位置 p , 那么返回 NO_SUCH_KEY 元素。否则, 如果找到关键字为 k 的位置 p (在底层), 那么利用 above 操作爬到 S 中这个数据项所在塔的位置 p 开始处, 很容易访问这些位置, 并删除 p 上方的所有位置。删除算法如图 3-42 中的说明, 详细描述留作习题 (R-3.19)。正如将在下一小节所看到的那样, 跳跃表中删除操作的运行时间的期望值为 $O(\log n)$ 。

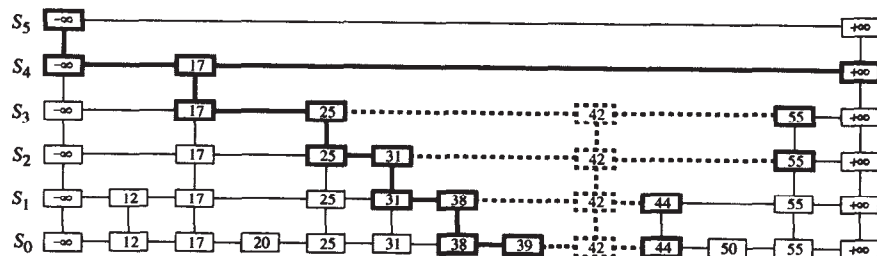


图3-41 将关键字为42的元素插入图3-39的跳跃表中。访问的位置和遍历的链接用粗线表示。用于存放新数据项的插入位置用虚线表示

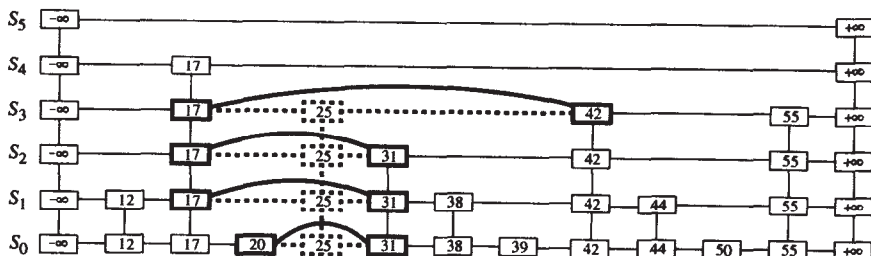


图3-42 从图3-41的跳跃表中删除关键字为25的数据项。初始查找之后访问的位置和遍历的链接用粗线表示，删除的位置用虚线表示

进行分析之前, 对将要讨论的跳跃表数据结构做一些微小改进。首先, 不需存储指向0层以上数据项的引用, 因为这些层需要的是对关键字的引用。其次, 实际上不需要 above 方法, 也不需要 before 方法。可以严格按照自顶向下、前向扫描的方式进行数据项的插入和删除。因此, 节省了“向上”和“向前”引用的空间。这种优化的细节留作习题 (C-3.26)。

3. 维持最顶层

跳跃表 S 必须将对 S 的最左上位置的引用维持为一个实例变量。而且对于想要在 S 的顶层后面继续插入新元素的任何插入必须具有一种策略。可以采取两种可能的动作, 每种动作各有其优点。

一种可能性是限制顶层 h 保持为某个固定值, 它是 n 的一个函数, 其中 n 为字典中的当前元素个数 (由分析可知, $h = \max\{10, 2\lceil \log n \rceil\}$ 是一个合理的选择, 取 $h = 3\lceil \log n \rceil$ 更安全)。实现这个选择意味着: 必须修改插入算法, 使得一旦到达最顶层 (除非 $\lceil \log n \rceil < \lceil \log(n+1) \rceil$), 插入过程即可停止。在这种情况下, 由于高度界限在增加, 至少可以多行进一层。

另一种可能性是继续进行插入新元素的过程, 只要保持随机数生成器返回的头结点即可。正如在跳跃表的分析中所看到的那样, 一次插入行进至超过 $O(\log n)$ 层的概率很小, 因此这种设计也是可行的。

然而, 不论哪一种选择, 其所导致的查找、插入和删除的期望时间均为 $O(\log n)$, 下一节中将证明这个结论。

3.5.3 跳跃表的概率分析

如上所示,跳跃表为有序字典提供了一种简单实现。但是,就最坏情况下的性能而言,跳跃表并不是一种出众的数据结构。事实上,如果对超过当前最高层的插入过程不设防,那么插入算法几乎会进入无限循环(它实际上不是无限循环,而是因为投掷均匀硬币重复出现正面的概率永远为0)。此外,不可能无限地向表中添加元素,而最终不会发生用完内存的情况。在任何情况下,如果在最高层 h 终止数据项插入,那么在有 n 个数据项、高为 h 的跳跃表 S 中进行findElement、insertItem和removeElement操作,最坏情况下的运行时间为 $O(n + h)$ 。当每个数据项所在的塔到达 $h-1$ 层时,就会出现这种最坏情况下的性能,其中 h 为 S 的高度。然而,发生这种事件的概率很小。用最坏情况下的性能作为度量标准,可以得出结论:跳跃表数据结构绝对次于本章早先讨论过的字典的其他实现方法。但是这并不是一个公平的分析,因为通常会粗略地高估最坏情况下的行为。

因为插入步骤包括随机化过程,对跳跃表更真实的分析涉及一些概率知识。首先,这好像只是一种说法,因为完整和深入的概率分析需要深入的数学知识。幸运的是,这样一个分析不必理解跳跃表的期望渐近行为。以下给出的非形式和直观的概率分析仅利用了概率论中的基本概念。

200

1. 限制跳跃表的高度

首先确定 S 的高度期望值 h (假设不会过早终止插入过程)。给定数据项存储在第 i 层某个位置的概率等于投掷一枚硬币时连续得到 i 次正面的概率,即为 $1/2^i$ 。因此,第 i 层至少有一个数据项的概率 P_i 至多为

$$P_i \leq n/2^i$$

对于 n 个不同的事件,出现其中任何一个事件的概率至多为出现每个事件的概率之和。

S 的高度 h 大于 i 的概率等于第 i 层至少有一个数据项的概率,即不超过 P_i 。这意味着 h 大于 $3\log n$ 的概率至多为

$$P_{3\log n} \leq n/(2^{3\log n}) = n/n^3 = 1/n^2$$

更一般地讲,给定一个常数 $c > 1$, h 大于 $c\log n$ 的概率至多为 $1/n^{c-1}$,即 h 小于或等于 $c\log n$ 的概率至少为 $1-1/n^{c-1}$ 。因此, S 的高度为 $O(\log n)$ 的概率较高。

2. 跳跃表中的查找时间分析

考虑在跳跃表 S 中进行查找的运行时间,请记住这样一次查找包括两个嵌套的while循环。只要某层的下一个关键字不大于查找关键字 k ,内层循环就在该层上进行前向扫描,外层循环则下移一层,并重复前向扫描迭代过程。因为 S 的高度 h 为 $O(\log n)$ 的概率较高,则向下的步骤数为 $O(\log n)$ 的概率也较高。

于是,还必须限定所进行的前向扫描步骤数。令 n_i 是在第 i 层进行前向扫描时检查的关键字数。观察可知,在起始位置的关键字之后,第 i 层中前向扫描所检查的每个其他关键字不可能属于第 $i+1$ 层。如果这些数据项中的任何一个在上一层上,则在上一个前向扫描步骤中会遇到它们。因此,在 n_i 中统计任一关键字的概率为 $1/2$ 。于是, n_i 的期望值恰好等于在出现正面时必须投掷一枚均匀硬币的期望次数。用 e 表示这个数量,则有

$$e = 1/2 \cdot 1 + 1/2 \cdot (1 + e)$$

因此, $e = 2$,且在任意第 i 层前向扫描所花费的期望时间为 $O(1)$ 。因为 S 具有 $O(\log n)$ 个层的概率较高,因而, S 中进行查找的期望时间为 $O(\log n)$ 。由类似分析可知,可以证明插入或删除的期望运行时间为 $O(\log n)$ 。

201

3. 跳跃表使用的空间

最后，分析跳跃表 S 所需的存储空间。如上所见，第 i 层期望的数据项数为 $n/2^i$ 。这表明 S 中期望的数据项总数为

$$\sum_{i=0}^h \frac{n}{2^i} = n \sum_{i=0}^h \frac{1}{2^i} < 2n$$

因此， S 的期望空间需求为 $O(n)$ 。

表3-5概括了跳跃表实现的字典的性能。

表3-5 跳跃表实现的字典的性能。 n 表示进行操作时字典中的数据项数。操作findAllElements和removeAllElements返回的迭代器大小为 s 。期望空间需求为 $O(n)$

操 作	时 间
keys, elements	$O(n)$
findElement, insertItem, removeElement	$O(\log n)$ (期望)
findAllElements, removeAllElements	$O(\log n + s)$ (期望)

3.6 Java 示例：AVL 树和红黑树

在这一节里，描述一种常规的二叉查找树类BinarySearchTree，以及如何扩展它以产生一种AVL树实现或者红黑树实现。BinarySearchTree类将类Item（见代码段3-1）的关键字-元素对作为元素，存储在其底层二叉树的位置（结点）中。代码段3-2显示了BinarySearchTree的代码。注意，底层二叉树 T 仅通过BinaryTree接口使用。假设BinaryTree还包括方法expandExternal和removeAboveExternal（见2.3.4节）。因此，类BinarySearchTree利用了代码重用。

findElement、insertItem和removeElement调用基于TreeSearch算法的辅助方法findPosition。实例变量actionPos存储最近查找、插入或删除结束时的位置。这个实例变量对于一棵二叉查找树的实现不是必需的，但可用于对BinarySearchTree（见代码段3-4~3-5和代码段3-8~3-9）进行扩展，使其能够确定上次查找、插入和删除所发生的位置。如果位置actionPos恰好在执行方法findElement、insertItem和removeElement后使用，则它具有预期的意义。

202

代码段3-1 用于字典中存储的关键字-元素对的类

```
public class Item {
    private Object key, elem;
    protected Item (Object k, Object e) {
        key = k;
        elem = e;
    }
    public Object key() { return key; }
    public Object element() { return elem; }
    public void setKey(Object k) { key = k; }
    public void setElement(Object e) { elem = e; }
}
```

代码段3-2 类BinarySearchTree

```
/** Realization of a dictionary by means of a binary search tree */
public class BinarySearchTree implements Dictionary {
    Comparator C; // comparator
    BinaryTree T; // binary tree
```

```

protected Position actionPos; // insertion position or parent of removed position

public BinarySearchTree(Comparator c) {
    C = c;
    T = (BinaryTree) new NodeBinaryTree();
}

// auxiliary methods:
/** Extract the key of the item at a given node of the tree. */
protected Object key(Position position) {
    return ((Item) position.element()).key();
}
/** Extract the element of the item at a given node of the tree. */
protected Object element(Position position) {
    return ((Item) position.element()).element();
}
/** Check whether a given key is valid. */
protected void checkKey(Object key) throws InvalidKeyException {
    if (!C.isComparable(key))
        throw new InvalidKeyException("Key "+key+" is not comparable");
}

/** Auxiliary method used by removeElement. */
protected void swap(Position swapPos, Position remPos){
    T.replaceElement(swapPos, remPos.element());
}

/** Auxiliary method used by findElement, insertItem, and removeElement. */
protected Position findPosition(Object key, Position pos) {
    if (T.isExternal(pos))
        return pos; // key not found and external node reached returned
    else {
        Object curKey = key(pos);
        if (C.isLessThan(key, curKey))
            return findPosition(key, T.leftChild(pos));
        else if (C.isGreaterThan(key, curKey)) // search in left subtree
            return findPosition(key, T.rightChild(pos)); // search in right subtree
        else
            return pos; // return internal node where key is found
    }
}

// methods of the dictionary ADT
public int size() {
    return (T.size() - 1) / 2;
}

public boolean isEmpty() {
    return T.size() == 1;
}

public Object findElement(Object key) throws InvalidKeyException {
    checkKey(key); // may throw an InvalidKeyException
    Position curPos = findPosition(key, T.root());
    actionPos = curPos; // node where the search ended
    if (T.isInternal(curPos))
        return element(curPos);
    else
        return NO_SUCH_KEY;
}

```

203

204

```

public void insertItem(Object key, Object element)
    throws InvalidKeyException {
    checkKey(key); // may throw an InvalidKeyException
    Position insPos = T.root();
    do {
        insPos = findPosition(key, insPos);
        if (T.isExternal(insPos))
            break;
        else // the key already exists
            insPos = T.rightChild(insPos);
    } while (true);
    T.expandExternal(insPos);
    Item newItem = new Item(key, element);
    T.replaceElement(insPos, newItem);
    actionPos = insPos; // node where the new item was inserted
}

public Object removeElement(Object key) throws InvalidKeyException {
    Object toReturn;
    checkKey(key); // may throw an InvalidKeyException
    Position remPos = findPosition(key, T.root());
    if (T.isExternal(remPos)) {
        actionPos = remPos; // node where the search ended unsuccessfully
        return NO_SUCH_KEY;
    }
    else {
        toReturn = element(remPos); // element to be returned
        if (T.isExternal(T.leftChild(remPos)))
            remPos = T.leftChild(remPos);
        else if (T.isExternal(T.rightChild(remPos)))
            remPos = T.rightChild(remPos);
        else { // key is at a node with internal children
            Position swapPos = remPos; // find node for swapping items
            remPos = T.rightChild(swapPos);
            do
                remPos = T.leftChild(remPos);
            while (T.isInternal(remPos));
            swap(swapPos, T.parent(remPos));
        }
        actionPos = T.sibling(remPos); // sibling of the leaf to be removed
        T.removeAboveExternal(remPos);
        return toReturn;
    }
}
}
}

```

205

3.6.1 AVL 树的 Java 实现

我们转向利用有 n 个内部结点的AVL树 T 实现有 n 个数据项的有序字典的实现细节及分析上来。对于 T 的插入和删除算法要求能够进行三结点重构，并确定两个兄弟结点之间的高度差。至于重构，应该通过添加方法`restructure`，扩展二叉树ADT的操作集合。容易看到，如果利用链表结构实现树 T ，那么`restructure`操作的执行时间为 $O(1)$ ，对于向量则不是这样（2.3.4节）。倾向于用链表表示一棵AVL树。

至于高度信息，可以将每个内部结点 v 的高度显式地存储在结点自身中。此外，还可在结点 v 处存储它的平衡因子。平衡因子定义为 v 的左子结点高度与右子结点高度之差。因此， v 的平衡因

子总是-1、0或1，只不过在插入或删除过程中，可能暂时变为-2或2。在插入或删除过程中，将会有 $O(\log n)$ 个结点的高度和平衡因子受到影响，并且可以维持 $O(\log n)$ 时间。

在代码段3-3~3-5中，给出了基于AVL树的字典数据结构的Java实现。代码段3-3中显示的类AVLItem扩展了Item类，Item类可用于表示二叉查找树中的关键字-元素对数据项。它定义了另一个实例变量height，表示结点的高度。代码段3-4~3-5中给出了完整的类AVLTree，它对BinarySearchTree（代码段3-2）进行了扩展。AVLTree的构造函数首先执行超类的构造函数，然后将RestructurableNodeBinaryTree赋给T，它是一个实现二叉树ADT的类，此外，它还支持方法restructure进行三结点重构。类AVLTree从它的超类BinarySearchTree继承了方法size、isEmpty、findElement、findAllElements和removeAllElements，但是重写了方法insertItem和removeElement。

代码段3-3 实现AVL树中一个结点的类。树中结点的高度被存储为实例变量

```
public class AVLItem extends Item {
    int height;
    AVLItem(Object k, Object e, int h) {
        super(k, e);
        height = h;
    }
    public int height() { return height; }
    public int setHeight(int h) {
        int oldHeight = height;
        height = h;
        return oldHeight;
    }
}
```

代码段3-4 类AVLTree的构造函数和辅助方法

```
/** Realization of a dictionary by means of an AVL tree. */
public class AVLTree extends BinarySearchTree implements Dictionary {
    public AVLTree(Comparator c) {
        super(c);
        T = new RestructurableNodeBinaryTree();
    }
    private int height(Position p) {
        if (T.isExternal(p))
            return 0;
        else
            return ((AVLItem) p.element()).height();
    }
    private void setHeight(Position p) { // called only if p is internal
        ((AVLItem) p.element()).setHeight(1+Math.max(height(T.leftChild(p)),
            height(T.rightChild(p))));
    }
    private boolean isBalanced(Position p) {
        // test whether node p has balance factor between -1 and 1
        int bf = height(T.leftChild(p)) - height(T.rightChild(p));
        return ((-1 <= bf) && (bf <= 1));
    }
    private Position tallerChild(Position p) {
        // return a child of p with height no smaller than that of the other child
        if (height(T.leftChild(p)) >= height(T.rightChild(p)))
```

```

        return T.leftChild(p);
    else
        return T.rightChild(p);
}

```

方法insertItem（代码段3-5）首先调用超类中的insertItem方法，它插入新的数据项，并将插入位置（图3-8中存储关键字54的结点）赋给实例变量actionPos。接着用辅助方法rebalance遍历从插入位置到根的路径。这个遍历会更新访问的所有结点的高度，如果需要则进行三结点重构。类似地，方法removeElement（代码段3-5）首先调用超类中的removeElement方法，执行数据项的删除，并将代替被删除位置的位置赋给实例变量actionPos，接着用辅助方法rebalance遍历从删除位置到根的路径。

代码段3-5 辅助方法rebalance以及类AVLTree的方法insertItem和removeElement

```

/**
 * Auxiliary method called by insertItem and removeElement.
 * Traverses the path of T from the given node to the root. For
 * each node zPos encountered, recomputes the height of zPos and
 * performs a trinode restructuring if zPos is unbalanced.
 */
private void rebalance(Position zPos) {
    while (!T.isRoot(zPos)) {
        zPos = T.parent(zPos);
        setHeight(zPos);
        if (!isBalanced(zPos)) {
            // perform a trinode restructuring
            Position xPos = tallerChild(tallerChild(zPos));
            zPos = ((RestructurableNodeBinaryTree) T).restructure(xPos);
            setHeight(T.leftChild(zPos));
            setHeight(T.rightChild(zPos));
            setHeight(zPos);
        }
    }
}

// methods of the dictionary ADT

/** Overrides the corresponding method of the parent class. */
public void insertItem(Object key, Object element)
    throws InvalidKeyException {
    super.insertItem(key, element); // may throw an InvalidKeyException
    Position zPos = actionPos; // start at the insertion position
    T.replaceElement(zPos, new AVLItem(key, element, 1));
    rebalance(zPos);
}

/** Overrides the corresponding method of the parent class. */
public Object removeElement(Object key)
    throws InvalidKeyException {
    Object toReturn = super.removeElement(key);
    // may throw an InvalidKeyException
    if (toReturn != NO_SUCH_KEY) {
        Position zPos = actionPos; // start at the removal position
        rebalance(zPos);
    }
}

```

```

    return toReturn;
}

```

208

3.6.2 红黑树的 Java 实现

在代码段3-6~3-9中, 给出了基于红黑树的字典数据结构的Java实现的某些部分。代码段3-6中所示的类RBItem对类Item进行了扩展, Item类可用于表示二叉查找树中的关键字-元素对数据项。它还定义了表示结点颜色的实例变量isRed, 以及用于设置返回颜色的方法。

代码段3-6 实现红黑树结点的类

```

public class RBItem extends Item {
    private boolean isRed;
    public RBItem(Object k, Object elem, boolean color) {
        super(k, elem);
        isRed = color;
    }
    public boolean isRed() {return isRed;}
    public void makeRed() {isRed = true;}
    public void makeBlack() {isRed = false;}
    public void setColor(boolean color) {isRed = color;}
}

```

代码段3-7 类RBTree的实例变量和构造函数

```

/** Realization of a dictionary by means of a red-black tree. */
public class RBTree extends BinarySearchTree implements Dictionary {
    static boolean Red = true;
    static boolean Black = false;

    public RBTree(Comparator C) {
        super(C);
        T = new RestructurableNodeBinaryTree();
    }
}

```

代码段3-7~3-9中显示了类RBTree的一部分, 这个类对BinarySearchTree(代码段3-2)进行了扩展。正如在类AVLTree中那样, RBTree的构造函数首先执行超类的构造函数, 然后将RestructurableNodeBinaryTree赋给T, 它是实现二叉树ADT的一个类, 此外, 支持方法restructure进行三结点重构(旋转)。类RBTree从BinarySearchTree继承了方法size、isEmpty、findElement、findAllElements和removeAllElements, 但是重写了方法insertItem和removeElement。类RBTree的几个辅助方法未显示。

方法insertItem(代码段3-8)和removeElement(代码段3-9)首先调用超类中的相应方法, 然后通过调用辅助方法, 沿着从更新位置(通过从超类继承的实例变量actionPos给出)到根的路径上进行旋转, 完成树的再平衡过程。

209

代码段3-8 字典方法insertItem和类RBTree的辅助方法remedyDoubleRed

```

public void insertItem(Object key, Object element)
    throws InvalidKeyException {
    super.insertItem(key, element); // may throw an InvalidKeyException
    Position posZ = actionPos; // start at the insertion position
    T.replaceElement(posZ, new RBItem(key, element, Red));
    if (T.isRoot(posZ))

```



```

        setBlack(posZ);
    else
        remedyDoubleRed(posZ);
}

protected void remedyDoubleRed(Position posZ) {
    Position posV = T.parent(posZ);
    if (T.isRoot(posV))
        return;
    if (!isPosRed(posV))
        return;
    // we have a double red: posZ and posV
    if (!isPosRed(T.sibling(posV))) { // Case 1: trinode restructuring
        posV = ((RestructurableNodeBinaryTree) T).restructure(posZ);
        setBlack(posV);
        setRed(T.leftChild(posV));
        setRed(T.rightChild(posV));
    }
    else { // Case 2: recoloring
        setBlack(posV);
        setBlack(T.sibling(posV));
        Position posU = T.parent(posV);
        if (T.isRoot(posU))
            return;
        setRed(posU);
        remedyDoubleRed(posU);
    }
}
}

```

210

代码段3-9 方法removeElement及其辅助方法

```

public Object removeElement(Object key) throws InvalidKeyException {
    Object toReturn = super.removeElement(key);
    Position posR = actionPos;
    if (toReturn != NO_SUCH_KEY) {
        if (wasParentRed(posR) || T.isRoot(posR) || isPosRed(posR))
            setBlack(posR);
        else
            remedyDoubleBlack(posR);
    }
    return toReturn;
}

protected void remedyDoubleBlack(Position posR) {
    Position posX, posY, posZ;
    boolean oldColor;
    posX = T.parent(posR);
    posY = T.sibling(posR);
    if (!isPosRed(posY)) {
        posZ = redChild(posY);
        if (hasRedChild(posY)) { // Case 1: trinode restructuring
            oldColor = isPosRed(posX);
            posZ = ((RestructurableNodeBinaryTree) T).restructure(posZ);
            setColor(posZ, oldColor);
            setBlack(posR);
            setBlack(T.leftChild(posZ));
            setBlack(T.rightChild(posZ));
            return;
        }
        setBlack(posR);
    }
}

```

```

    setRed(posY);
    if (!isPosRed(posX)) { // Case 2: recoloring
        if (!T.isRoot(posX))
            remedyDoubleBlack(posX);
        return;
    }
    setBlack(posX);
    return;
} // Case 3: adjustment
if (posY == T.rightChild(posX))
    posZ = T.rightChild(posY);
else
    posZ = T.leftChild(posY);
((RestructurableNodeBinaryTree)T).restructure(posZ);
setBlack(posY);
setRed(posX);
remedyDoubleBlack(posR);
}

```

211

3.7 习题

基础题

- R-3.1 在初始为空的二叉查找树中，插入具有如下关键字的数据项：30, 40, 24, 58, 48, 26, 11, 13。画出每次插入后的树。
- R-3.2 Amongus教授声称，将一组固定的元素插入一棵二叉查找树中的次序是无关紧要的，每次插入后的树都是一样的。给出一个例子，证明Amongus教授的说法是错误的。
- R-3.3 Amongus教授声称，将上题声明打个补丁，即将一组固定的元素插入一棵AVL树中的次序是无关紧要的，每次插入后的AVL树是一样的。给出一个例子，证明Amongus教授的说法仍是错误的。
- R-3.4 图3-8中所做的旋转是单旋还是双旋？图3-10中的旋转呢？
- R-3.5 画出将关键字为52的数据项插入图3-10b的AVL树后得到的AVL树。
- R-3.6 画出将关键字为62的数据项从图3-10b的AVL树中删除后得到的AVL树。
- R-3.7 解释为什么在用序列表示的 n 个结点的二叉树中进行旋转所需的时间为 $\Omega(n)$ 。
- R-3.8 图3-12a的多路查找树是一棵(2, 4)树吗？证明你的结论。
- R-3.9 对一棵(2, 4)树中的结点 v 进行分裂，另一种方法是将 v 划分成 v' 和 v'' ，其中 v' 是一个2-结点， v'' 是一个3-结点。在这种情况下，在 v 的父结点中存储关键字 k_1 、 k_2 、 k_3 或 k_4 中的哪一个？为什么？
- R-3.10 Amongus教授声称，存储数据项集的一棵(2, 4)树的结构总是相同，与数据项被插入的次序无关。证明Amongus教授是错误的。
- R-3.11 考虑下列关键字序列：

(5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1)

按照给定的次序，考虑插入具有这组关键字的数据项：

- 向初始为空的(2, 4)树 T' 中进行插入。
- 向初始为空的红黑树 T'' 中进行插入。

画出每次插入后的树 T' 和 T'' 。

- R-3.12 利用本章描述的对对应关系规则，画出对应同一棵(2, 4)树的4棵不同的红黑树。
- R-3.13 画出一棵不是AVL树的红黑树。你的树至少有6个结点，但不超过16个。
- R-3.14 对于以下关于红黑树的一些陈述，确定正误。如果你认为正确，进行证明，否则给出反例。
- 红黑树的子树本身也是红黑树。
 - 外部结点的兄弟结点或者为外部结点，或者为红色。
 - 给定一棵红黑树 T ，存在关联 T 的唯一(2, 4)树 T' 。
 - 给定一棵(2, 4)树 T ，存在关联 T 的唯一红黑树 T' 。

212

- R-3.15 在初始为空的伸展树中, 执行如下操作序列, 并画出每次操作后的树。
- 按照所给次序, 插入关键字0, 2, 4, 6, 8, 10, 12, 14, 16, 18。
 - 按照所给次序, 查找关键字1, 3, 5, 7, 9, 11, 13, 15, 17, 19。
 - 按照所给次序, 删除关键字0, 2, 4, 6, 8, 10, 12, 14, 16, 18。
- R-3.16 如果按照关键字递增的次序访问一棵伸展树的数据项, 这棵伸展树看起来像什么?
- R-3.17 在伸展树中进行zig-zig、zig-zag和zig更新操作, 需要多少次三结点重构? 利用图形解释你的计数。
- R-3.18 画出在图3-42所示的跳跃表上执行如下操作序列的结果: `removeElement(38)`、`insertItem(48, x)`、`insertItem(24, y)`、`removeElement(55)`。假设为首次插入进行投币的结果为两个正面后接一个反面, 为第二次插入进行投币的结果为三个正面后接一个反面。
- R-3.19 给出`removeElement`字典操作的伪代码描述, 假设用跳跃表结构实现字典。

创新题

- C-3.1 给定两个各有 n 个数据项的有序字典 S 和 T , 并用基于数组的有序序列实现 S 和 T 。描述在 S 和 T 的关键字的并集中找出第 k 个最小关键字的 $O(\log n)$ 时间的算法 (假设没有重复元素)。
- C-3.2 设计一个在基于有序数组实现的有序字典上执行操作`findAllElements(k)`的算法, 并证明其运行时间为 $O(\log n + s)$, 其中 n 为字典中的元素个数, s 是返回的数据项数。
- C-3.3 设计一个在基于二叉查找树实现的有序字典上执行操作`findAllElements(k)`的算法, 并证明其运行时间为 $O(h + s)$, 其中 h 为树高, s 是返回的数据项数。
- C-3.4 描述如何在基于二叉查找树实现的有序字典上执行操作`removeAllElements(k)`的算法, 并证明其运行时间为 $O(h + s)$, 其中 h 为树高, s 是返回的迭代器大小。
- C-3.5 画出一棵AVL树, 满足单个`removeElement`操作从叶结点到根所需的三结点重构 (或旋转) 为 $\Theta(\log n)$ 次, 以便恢复高度平衡性质 (利用三角形表示不受这个操作影响的子树)。
- C-3.6 指明在一个AVL树实现的字典中, 如何用 $O(s \log n)$ 时间执行操作`removeAllElements(k)`, 其中 n 是操作执行时字典中的元素个数, s 是操作返回的迭代器大小。
- C-3.7 如果维持对一棵AVL树中最左边位置的引用, 那么操作`first`可以在 $O(1)$ 时间内执行完成。描述在其他字典方法的实现中, 需要如何进行修改, 使其维持对最左边位置的引用。
- C-3.8 证明利用 $O(n)$ 次旋转, 可将任何一棵 n 个结点的二叉树转换成任何其他一棵 n 个结点的二叉树。
提示: 证明 $O(n)$ 次旋转足以将任何一棵二叉树转换成为左链 (left chain), 其中每个内部结点都有一个外部右子结点。
- C-3.9 证明对于在AVL树中执行`expandExternal`操作之后变成不平衡的那些结点, 在`insertItem`操作执行期间, 它们在从新插入结点到根的路径上可能不连续。
- C-3.10 设 D 是AVL树实现的有 n 个数据项的有序字典。证明如何用 $O(\log n + s)$ 的时间实现 D 上的以下操作, 其中 s 是返回的迭代器规模:
`findAllInRange(k_1, k_2)`: 返回 D 中关键字为 k 的所有元素的迭代器, 且满足 $k_1 \leq k \leq k_2$ 。
- C-3.11 设 D 是AVL树实现的有 n 个数据项的有序字典。证明如何用 $O(\log n)$ 的时间实现 D 上的下列方法:
`countAllInRange(k_1, k_2)`: 计算并返回 D 中关键字为 k 的数据项个数, 且满足 $k_1 \leq k \leq k_2$ 。
注意这个方法返回一个整数。
提示: 你需要对AVL树数据结构进行扩展, 并在每个内部结点中添加一个新的域, 同时提供在更新过程中维持这个域的方式。
- C-3.12 证明在AVL树中执行`removeAboveExternal`操作之后, 执行`removeElement`字典操作期间, 至多有一个结点成为不平衡结点。
- C-3.13 证明在AVL树中进行任何一次插入之后, 至多需要一次三结点重构操作 (对应于一个单旋或双旋操作), 即可恢复平衡性质。
- C-3.14 设 T 和 U 是分别存储 n 个和 m 个结点的(2, 4)树, 满足 T 中所有数据项的关键字小于 U 中所有数据项的关键字。描述一个 $O(\log n + \log m)$ 时间的方法, 将 T 和 U 连接成为一棵树, 这棵树存储 T 和 U 中的所有数据项 (销毁原来的 T 和 U)。

- C-3.15 对于红黑树 T 和 U ，重复上一个问题。
- C-3.16 证明定理3.3。
- C-3.17 用于将红黑树中的结点标记为“红”或“黑”的布尔值不是严格需要的。描述一种模式实现红黑树，而无需向二叉查找树结点中增加任何额外的空间。你的模式是如何影响红黑树中查找和更新操作的运行时间的？
- C-3.18 设 T 是一棵存储 n 个数据项的红黑树， k 是 T 中某个数据项的关键字。证明如何用 $O(\log n)$ 时间，由 T 构造两棵红黑树 T' 和 T'' ，满足 T' 中包含 T 中所有小于 k 的关键字， T'' 中包含 T 中所有大于 k 的关键字。这个操作将销毁 T 。
- C-3.19 可归并堆（mergeable heap）支持操作insert(k, x)、remove(k)、unionWith(h)和minElement()，其中unionWith(h)执行将可归并 h 与当前堆进行合并的操作，并销毁原来的两个堆。描述可归并堆ADT的一个具体实现，满足对于它进行的所有操作都会达到 $O(\log n)$ 的时间性能。为简单起见，你可以假设现有的可归并堆中的所有关键字互不相同，尽管严格意义上并不需要这样。
- C-3.20 考虑伸展树的一种变体，称为半伸展树（half-splay tree），它在伸展一个深度为 d 的结点时，在达到 $\lfloor d/2 \rfloor$ 深度的结点时即停止伸展。对半伸展树做一平摊分析。
- C-3.21 标准伸展步骤需要进行两遍，一遍是向下的过程，找出需要伸展的结点 x ，另一遍进行向上的过程，对结点 x 进行伸展。描述一个方法，用于在一遍向下的过程中伸展和查找 x 。每一子步现在需要考虑通向 x 的路径中的下两个结点，在最后可能执行zig子步。描述执行每个zig-zig、zig-zag和zig子步的细节。
- C-3.22 描述对 n 个结点的伸展树 T 的访问序列，其中 n 为奇数，使得 T 由单个内部结点的链组成，且外部结点作为子结点，满足沿 T 向下的内部结点路径在左子结点和右子结点之间交替出现。
- C-3.23 证明定理3.10。确定证明的一种方式，可以将结点大小重新定义为其子结点被访问次数之和。并证明定理3.8的证明过程仍然可行。
- C-3.24 给定数据项为 $(x_0, x_1, \dots, x_{n-1})$ 的序列 S ，满足 S 中的每个数据项 x_i 是一个已知权为 a_i 的正整数。设 A 表示 S 中所有元素的总权值。构造一个 $O(n \log n)$ 时间的算法，建立 S 的一棵二叉树 T ，满足每个数据项 a_i 的深度为 $O(\log A/a_i)$ 。
提示：找出具有最大 j 的数据项 x_j ，满足 $\sum_{i=0}^{j-1} a_i < A/2$ 。考虑把这个数据项放在根上，并对由此产生的两个子序列进行递归调用。
- C-3.25 设计上一个问题的线性时间算法。
- C-3.26 证明利用跳跃表有效实现字典时，实际上不需要方法above(p)和before(p)。也就是说，可以在一个跳跃表中，严格按照自顶向下、前向扫描的方式进行数据项的插入和删除操作，而不需用above或before方法。
- C-3.27 描述在有序序列实现的字典中，如何实现基于定位器的方法before(ℓ)，以及基于定位器的方法closestBefore(k)。对于利用无序序列实现的字典，上述的两个方法如何实现？这些方法的运行时间是多少？
- C-3.28 利用跳跃表实现字典，重复上一个问题。在你的实现中，这两个基于定位器的方法的期望运行时间是多少？
- C-3.29 假定 $n \times n$ 数组 A 的每一行由1和0组成，满足在 A 的任一行中，所有的1都在那一行的0之前。假设 A 已在内存中，描述一个运行时间为 $O(n \log n)$ （不是 $O(n^2)$ 时间）的方法，统计 A 中1的个数。
- C-3.30 描述一个存储 n 个元素的有效有序字典结构，这些元素关联的关键字有 $k < n$ 个，并且这组关键字取自一个全序集，即关键字集比元素个数小。你的结构对于所有有序字典操作的期望时间应为 $O(\log k + s)$ ，其中 s 是返回的元素个数。

程序设计

- P-3.1 利用AVL树、跳跃表或红黑树实现有序字典ADT的方法。
- P-3.2 提供跳跃表操作的图示动画。可视化插入期间数据项在跳跃表中向上移动的过程，以及删除期间数据项被链接出跳跃表的过程。

3.8 本章注记

Knuth[119]和Mehlhorn[148]的著作中全面论述了上面讨论的一些数据结构。AVL树由Adel'son-Vel'skii和Landis [2] 提出。二叉查找树的平均高度分析可在Aho、Hopcroft和Ullman[8]以及Cormen、Leiserson和Rivest[55]的著作中找到。Gonnet和Baeza-Yates的著作[81]中包含大量关于字典实现之间的理论和实验比较。Aho、Hopcroft和Ullman的著作[7]讨论了(2, 3)树, 它类似于(2, 4)树。Bayer[23]给出了红黑树的定义, 在Guibas和Sedgewick的著作[91]中对红黑树作了进一步讨论。Sleator和Tarjan[189] (同时见[200]) 发明了伸展树。其他一些材料可以在Mehlhorn[148]和Tarjan[200]的著作中找到。一些章节可在Mehlhorn和Tsakalidis的著作[152]中找到。Knuth的著作[119]是优秀的参考书, 包含了早期平衡树的一些方法。习题C-3.25受到Mehlhorn的论文[147]启发。跳跃表由Pugh[170]引入。对跳跃表的分析是对Motwani和Raghavan著作[157]中给出的相应内容的简化。对支持字典ADT的其他概率构造感兴趣的读者, 可参阅Motwani和Raghavan所著的教材[157]。至于对跳跃表的更深入的分析, 读者可参考其中关于跳跃表的论文[115, 163, 167]。



热力学第二定律表明，自然界有向无序发展的趋向。而人类却相反，喜欢有秩序。实际上，保持数据有序有几个优点。例如，二分搜索算法仅对有序数组或者向量是正确的。因为人们希望把计算机作为工具，本章研究排序算法及其应用。回顾排序问题，定义如下。设 S 是 n 个元素的序列，可按照全序关系对这些元素进行比较，即比较序列 S 中的任意两个元素的大小，或者看看它们是否相等。我们希望用以下方式重排 S ，使得元素按照增序排列（如果 S 中存在相等的元素，则按非降序排列）。

在前面几章里，已经给出了几个排序算法。特别是在2.4.2节，介绍了一种称为PQ-Sort的简单排序模式，它将元素插入优先队列，然后通过一系列removeMin操作，按照非降序提取它们。如果用序列实现优先队列，那么算法的运行时间为 $O(n^2)$ ，并且对应于插入排序或选择排序，这取决于是否保持序列有序。如果用堆（2.4.3节）实现优先队列，那么算法的运行时间为 $O(n \log n)$ ，相应的排序算法称为堆排序。

在这一章里，我们给出另外几个排序算法，以及它们基于的算法设计方法。其中的两个算法称为归并排序（merge-sort）和快速排序（quick-sort），基于分治法设计思想，它也广泛应用于其他问题中。还有两个排序算法称为桶排序（bucket-sort）和基数排序（radix-sort），基于散列法中使用的桶数组法。我们还引入了集合（set）抽象数据类型，并表明归并排序算法中使用的归并技术如何应用于其方法的实现中。集合抽象数据类型有一个重要的子类型，称为划分（partition），它支持的主要方法有union和find，同时划分有快得出奇的实现。我们还表明，可在 $O(n \log^* n)$ 时间内，执行 n 个union和find操作组成的序列，其中 $\log^* n$ 表示可以应用对数函数的次数，它从 n 开始，到1之前。这个分析提供了平摊分析的一个重要示例。

在本章中，我们还将讨论排序算法的下界证明方法，表明要对 n 个数进行排序，任何基于比较的方法所需的时间至少为 $\Omega(n \log n)$ 。但在某些情况下，我们对整个数据集合的排序不感兴趣，而只是对选择集合中的第 k 个最小元素感兴趣。事实上，可以证明这个选择（selection）问题比排序问题的求解可能要快得多。本章中包含的Java实现例子就是针对原位快速排序的。

贯穿本章，我们假设全序关系定义在待排序的元素上。如果这个关系由比较器（2.4.1节）导出，则假设一次比较所花的时间为 $O(1)$ 。

4.1 归并排序

在这一节里，我们介绍一种排序技术，称为归并排序，使用递归过程可以简单而又简洁地对

这种排序方法进行描述。

4.1.1 分治法

归并排序是一种基于分治法 (divide-and-conquer) 的算法设计方法。通常意义下, 分治法可由如下描述的三个步骤组成:

(1) 划分: 如果输入大小小于某一阈值 (threshold) (比如一个或两个元素), 则直接求解这个小问题, 并返回得到的解。否则, 将输入数据分成两个或多个不相交的子集。

(2) 递归: 递归求解子集关联的子问题。

(3) 求解: 取子问题的解, 并将它们“归并”成为原问题的解。

归并排序将分治技术应用于求解排序问题。

1. 用分治法进行排序

回忆一下在排序问题中, 给定 n 个通常存储在表、向量、数组或序列中的对象集合, 以及定义这些对象的全序关系的比较器。我们要产生这些对象的一个有序表示。为了具有一般性, 我们主要考虑将对象的序列 S 作为输入并返回 S 的有序序列这样的排序问题。如要将其特殊化为其他一些线性结构, 如表、向量或数组, 则非常直观, 将其留作习题 (R-4.3 和 R-4.13)。对于有 n 个元素的序列 S 的排序问题, 分治法的三个步骤如下:

(1) 划分: 如果 S 中有 0 个或 1 个元素, 直接返回 S ; 它已经有序。否则 (S 中至少有两个元素), 从 S 中删除所有元素, 并将它们分别放入两个序列 S_1 和 S_2 中, S_1 和 S_2 各包含大约 S 中的一半元素, 即 S_1 包含 S 中的前 $\lceil n/2 \rceil$ 个元素, S_2 包含 S 中的后 $\lfloor n/2 \rfloor$ 个元素。

(2) 递归: 递归求解子问题 S_1 和 S_2 。

219 (3) 求解: 归并有序序列 S_1 和 S_2 , 使它们成为一个有序序列, 并将其中的元素放回 S 中。

图 4-1 给出了归并算法的一个示意图。我们也可利用一棵二叉树 T (称为归并排序树 (merge-sort tree)) 说明归并排序算法的执行过程 (如图 4-2 所示)。 T 中的每个结点表示归并排序算法的一次递归调用 (或调用)。将 T 中的每个结点 v 与调用时处理的序列 S 关联起来。将结点 v 的子结点与递归调用 S 的子序列 S_1 和 S_2 关联起来。 T 的外部结点则关联 S 的各个元素, 对应于不再进行递归调用的算法的实例。

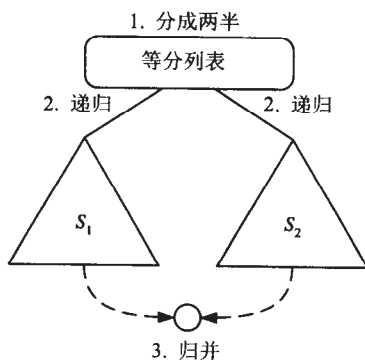


图4-1 归并算法的示意图

图 4-2 概括了归并算法的执行过程, 它显示了归并排序树中每个结点上处理的输入和输出序列。基于归并排序树的算法可视化过程可以帮助我们分析归并排序算法的运行时间。特别是在归并排序的每次递归调用中, 由于输入序列的大小大约减半, 则归并排序树的高度大约为 $\log n$ (回忆可知如果对数的底为 2, 则可将 n 省略)。

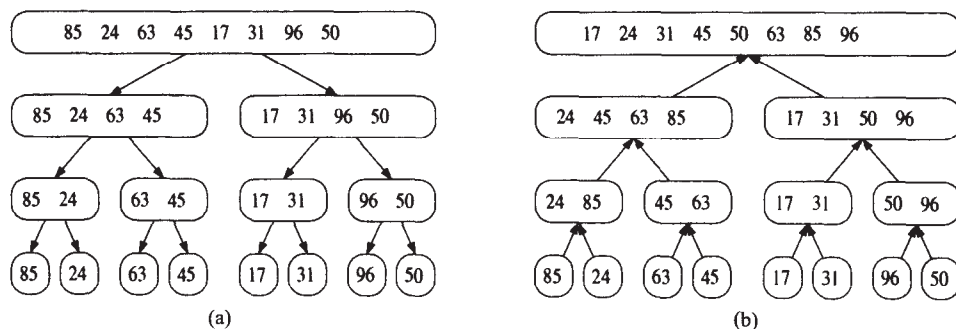


图4-2 用于在有8个元素的序列上的执行归并排序算法的归并排序树 T : (a) T 中每个结点上处理的输入序列; (b) T 中每个结点上产生的输出序列

220

关于划分步骤, 我们回忆符号 $\lceil x \rceil$ 表示 x 的向上取整 (ceiling), 即满足 $x \leq m$ 的最小整数 m 。类似地, 符号 $\lfloor x \rfloor$ 表示 x 的向下取整 (floor), 即满足 $k \leq x$ 的最大整数 k 。因此, 划分步骤尽可能地等分表 S , 结论如下:

定理4.1 在大小为 n 的序列上执行归并排序所关联的归并排序树的高度为 $\lceil \log n \rceil$ 。

定理4.1的证明留作习题 (R-4.1)。

给出归并排序的思想及其工作方式的说明后, 我们现在详细分析分治算法的每个步骤。归并排序算法的划分和递归步骤是简单的: 它在位序 $\lceil n/2 \rceil$ 处, 对 n 个元素的序列进行划分, 并将这些更小序列作为参数传递到递归调用的过程中。困难的一步是求解步骤, 它需要将两个有序序列归并成一个有序序列。

2. 归并两个有序序列

算法4-1中显示的算法merge将两个有序序列 S_1 和 S_2 进行归并, 其方法是: 算法merge不断地从这两个子序列中删除最小的元素, 并将其添加到输出序列 S 的末尾, 直到这两个序列中有一个为空。此时, 我们将另一个非空序列中的剩余元素复制到输出序列中。

算法4-1 归并两个有序序列的算法merge

算法 merge(S_1, S_2, S):

输入: 非降序排列的序列 S_1 和 S_2 , 以及一个空序列 S

输出: 包含 S_1 和 S_2 中的所有元素的序列 S , 这些元素按非降序排列, 同时 S_1 和 S_2 变为空序列

```
while (not ( $S_1$ .isEmpty() or  $S_2$ .isEmpty())) do
    if  $S_1$ .first().element()  $\leq$   $S_2$ .first().element() then
        {将 $S_1$ 中的第一个元素移到 $S$ 的末尾}
         $S$ .insertLast( $S_1$ .remove( $S_1$ .first()))
```

```
    else
        {将 $S_2$ 中的第一个元素移到 $S$ 的末尾}
         $S$ .insertLast( $S_2$ .remove( $S_2$ .first()))
```

```
{将 $S_1$ 中的剩余元素移到 $S$ 的末尾}
```

```
while (not  $S_1$ .isEmpty()) do
     $S$ .insertLast( $S_1$ .remove( $S_1$ .first()))
```

```
{将 $S_2$ 中的剩余元素移到 $S$ 的末尾}
```

```
while (not  $S_2$ .isEmpty()) do
     $S$ .insertLast( $S_2$ .remove( $S_2$ .first()))
```

221

首先做一些简单观察, 来分析merge算法的运行时间。设 n_1 和 n_2 分别表示 S_1 和 S_2 中的元素个数。假设序列 S_1 、 S_2 和 S 已被实现, 因此访问、插入和删除它们的第一个位置和最后一个位置的时间为 $O(1)$ 。这就是基于循环数组和双向链表(2.2.3节)实现的情况。算法merge有三个while循环。由于我们所做的假设, 每次在每个循环内进行的操作所需时间为 $O(1)$ 。关键的观察在于, 在其中的一个循环的每次迭代中, 要从 S_1 或 S_2 中删除一个元素。因为在 S_1 或 S_2 中不执行任何插入操作, 这就蕴涵着这三个循环的迭代总次数为 n_1+n_2 。因此, 算法merge的运行时间为 $O(n_1+n_2)$ 。概括如下:

定理4.2 归并两个有序序列 S_1 和 S_2 的时间为 $O(n_1+n_2)$, 其中 n_1 和 n_2 分别表示 S_1 和 S_2 的大小。

图4-3给出了算法merge的执行过程的示例。

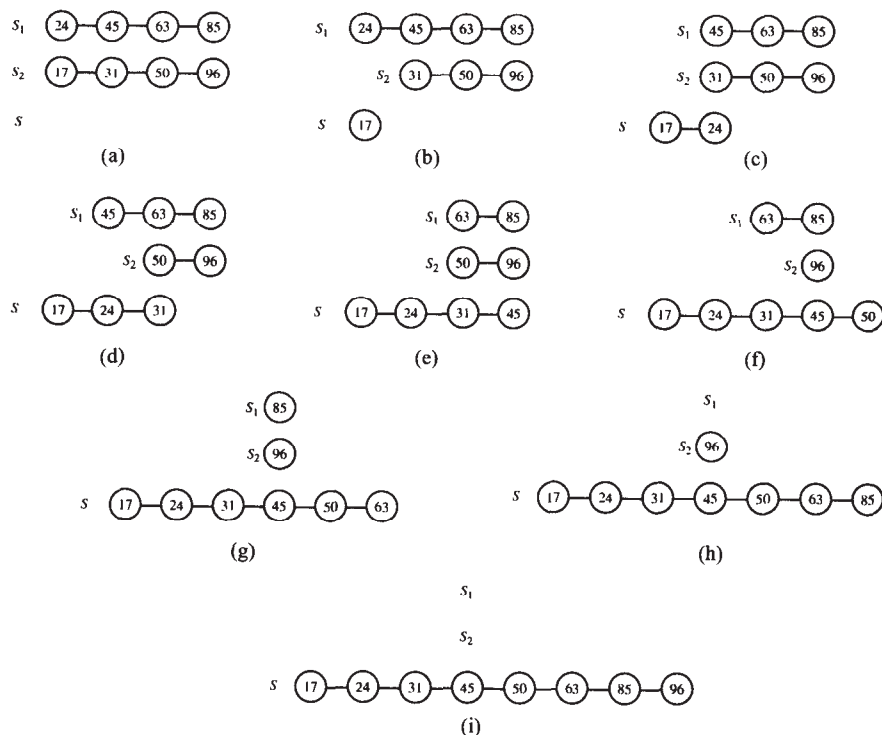


图4-3 算法merge的执行过程示例

3. 归并排序的运行时间

给出了归并排序算法的细节之后, 我们现在分析整个归并排序算法的运行时间, 假设给定 n 个元素的输入序列。为简单起见, 假设 n 是2的幂。如果 n 不是2的幂, 我们的结论仍然成立, 证明过程留作习题 (R-4.4)。

我们参考归并排序树 T , 分析归并排序算法。我们称关联结点 v 的递归调用的运行时间为 T 的结点 v 上所花费的时间 (time spent at a node v), 这不包括等待与 v 的子结点关联的递归调用终止所花费的时间。换句话说, 结点 v 上所花费的时间包括划分和求解的运行时间, 但不包含递归步骤的运行时间。观察可见, 划分步骤的细节是直观的; 这一步的运行时间与结点 v 上序列的大小成正比。同时, 如同在定理4.2中所表明的那样, 求解步骤由归并两个有序子序列组成, 也花费线性时间。也就是说, 设 i 表示结点 v 的深度, 结点 v 上花费的时间为 $O(n/2^i)$, 因为结点 v 上递归调用处理的序列大小为 $n/2^i$ 。

全面观察图4-4中的树 T , 可见, 给定“结点上所花费的时间”的定义, 归并排序的运行时间等于 T 中结点所花费的时间之和。观察可见, T 中深度 i 上正好有 2^i 个结点, 这蕴涵着 T 中深度 i 上的所有结点所花费的总时间为 $O(2^i \cdot n/2^i)$, 即为 $O(n)$ 。由定理4.1可知, T 的高度为 $\log n$ 。因此, T 的 $\log n + 1$ 个层的每一层上所花费的时间是 $O(n)$, 则有如下结论:

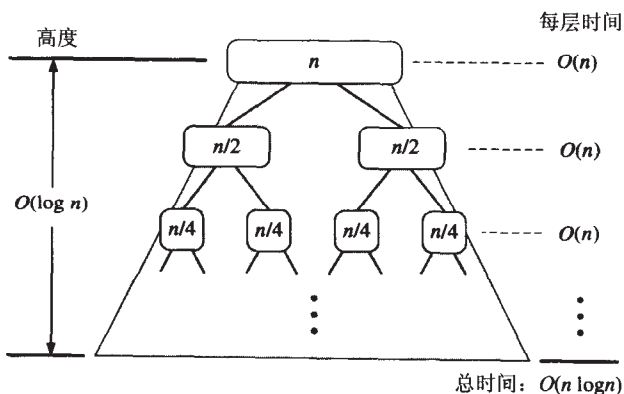


图4-4 归并排序运行时间的可视化分析。归并排序树的每个结点上标记出了其子问题的大小

定理4.3 归并排序最坏情况下的运行时间为 $O(n \log n)$ 。

223

4.1.2 归并排序和递归方程

存在另一种方法, 可以证明归并排序算法的运行时间为 $O(n \log n)$ 。设函数 $t(n)$ 表示归并排序在输入序列大小为 n 时最坏情况下的运行时间。因为归并排序是递归的, 我们可以利用以下方程表征函数 $t(n)$, 其中 $t(n)$ 根据自身递归表示如下:

$$t(n) = \begin{cases} b & \text{如果 } n=1 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & \text{其他情况} \end{cases}$$

其中 $b > 0$ 和 $c > 0$ 是常数。上述函数的表征方式称为递归方程 (recurrence equation) (见1.1.4节和5.2.1节), 这是因为函数出现在方程的左、右两边。尽管这种表征是正确的和准确的, 但是我们确实想要的是不包含 $t(n)$ 自身对 $t(n)$ 的大 O 表征形式 (即我们想要 $t(n)$ 的封闭形式的表征)。

为了提供 $t(n)$ 的封闭形式的表征, 我们仅关注 n 为2的幂的情况, 对于更一般的情况, 我们对 $t(n)$ 的渐近表征仍然成立, 把这个问题留作习题 (R-4.4)。在这种情况下, 可将 $t(n)$ 的定义简化为

$$t(n) = \begin{cases} b & \text{如果 } n=1 \\ 2t(n/2) + cn & \text{其他情况} \end{cases}$$

但是, 即使如此, 我们仍然需要尝试以封闭形式表征这个递归方程。一种方式是反复应用这个方程, 假设 n 相对较大。例如多次应用这个方程以后, 我们可以编写一个关于 $t(n)$ 的新的递归方程, 如下:

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2 t(n/2^2) + 2cn \end{aligned}$$

如果再次应用方程, 我们得到

$$t(n) = 2^3 t(n/2^3) + 3cn$$

应用方程 i 次之后, 则得到

$$t(n) = 2^i t(n/2^i) + icn$$

剩下的问题是确定何时停止这个过程。为了明确何时停止，回忆当 $n = 1$ 时，转换到闭合形式 $t(n) = b$ ，当 $2^i = n$ 时停止。即 $i = \log n$ 时停止。进行这种代换，得

$$\begin{aligned} t(n) &= 2^{\log n} t(n/2^{\log n}) + (\log n) cn \\ &= nt(1) + cn \log n \\ &= nb + cn \log n \end{aligned}$$

224 即我们得到 $t(n)$ 为 $O(n \log n)$ 这个事实的另一种证明方法。

4.2 集合抽象数据类型

在这一节里，我们引入集合（set）ADT。集合是不同对象的容器，即在一个集合中不存在重复的元素，也没有关键字甚至次序的明确概念。即使如此，我们仍然在排序这一章中讨论集合问题，这是因为在有效实现集合ADT的操作时，排序起着重要作用。

1. 集合及其某些用途

首先，回忆两个集合 A 和 B 的并（union）、交（intersection）和差（subtraction）的数学定义：

$$\begin{aligned} A \cup B &= \{x: x \in A \text{ 或 } x \in B\} \\ A \cap B &= \{x: x \in A \text{ 且 } x \in B\} \\ A - B &= \{x: x \in A \text{ 且 } x \notin B\} \end{aligned}$$

示例4.1 大多数因特网搜索引擎为其字典数据库中的每个字 x ，存储包含 x 的Web页面的集合 $W(x)$ ，并且用唯一因特网地址标识每个Web页面。当给出查询字 x 时，这样的搜索引擎只需返回集合 $W(x)$ 中的Web页面，这些页面按照某种专有的优先级排列，而优先级的计算基于页面“重要性”。但是，当给出两个查询字 x 和 y 时，这样的搜索引擎必定首先计算 $W(x) \cap W(y)$ 的交集，然后按照优先级排列顺序返回得到的集合中的Web页面。多种搜索引擎利用了本节为这个计算描述的集合交集算法。

2. 集合ADT中的基本方法

作用在集合 A 上的集合ADT的基本方法如下：

- **union(B)**：用 A 与 B 的并集代替 A ，即执行 $A \leftarrow A \cup B$ 。
- **intersect(B)**：用 A 与 B 的交集代替 A ，即执行 $A \leftarrow A \cap B$ 。
- **subtract(B)**：用 A 与 B 的差代替 A ，即执行 $A \leftarrow A - B$ 。

上面定义了union操作、intersect操作和subtract操作，这些操作对集合 A 中的内容进行修改。

225 另一方面，我们也可以定义这些方法，使得不对 A 进行修改，而是返回一个新的集合。

4.2.1 简单的集合实现

实现集合最简单的方式之一是，将它的元素存储在有序序列中。例如，在一些针对泛型数据结构的软件库中包括有这种实现。因此，考虑用有序序列实现集合ADT（在多个习题中考虑了其他的实现）。假如所有集合使用相同的次序，则在集合中的元素之间可使用任何一致的全序关系。

我们利用归并算法的泛型版本实现集合的三个基本操作。这个归并算法取表示输入集合的两个有序序列作为输入，构造一个表示输入集合的并、交或差的输出集合的序列。

泛型归并算法分别反复考察和比较输入序列 A 和 B 中的当前元素 a 和 b ，并查明是否 $a < b$ 、 $a = b$ 或 $a > b$ 。然后，基于比较的结果，确定是否将 a 或 b 中的一个元素复制到输出序列 C 的末尾。这

个确定是基于所进行的特定操作是并、交，还是差做出的。然后考虑一个序列或是两个序列的下一个元素。例如，在并操作中，进行如下过程：

- 如果 $a < b$ ，我们将 a 复制到输出序列中，并考察 A 中的下一个元素。
- 如果 $a = b$ ，我们将 a 复制到输出序列中，并考察 A 和 B 中的下一个元素。
- 如果 $a > b$ ，我们将 b 复制到输出序列中，并考察 B 中的下一个元素。

泛型归并的性能

我们现在分析泛型归并算法的运行时间。在每次迭代过程中，比较输入序列 A 和 B 中的两个元素，并可能将其中的一个元素复制到输出序列中，继续考察 A 或 B 或者这两个序列中的下一个元素。假设比较和复制元素的时间为 $O(1)$ ，总运行时间为 $O(n_A + n_B)$ ，其中 n_A 是 A 的大小， n_B 是 B 的大小，即泛型归并所需时间与涉及的元素个数成正比。因此，可得以下定理：

定理4.4 可用有序序列以及支持操作 **union**、**intersect** 和 **subtract** 的泛型归并模式实现集合 ADT，这些操作的运行时间为 $O(n)$ ，其中 n 为所涉及的集合大小之和。

226

还有一种集合 ADT 的特殊且重要的版本，它只用于处理不相交集的集合。

4.2.2 具有 union-find 操作的划分

划分是一个不相交集的集合。我们利用定位器（2.4.4 节）定义划分 ADT 的方法，来访问存储在集合中的元素。在这种情况下，每个定位器就像一个指针，若元素存储在我们的划分中，则定位器提供对位置（结点）的快速访问。

- **makeSet(e)**: 创建包含元素 e 的一个集合，并返回元素 e 的定位器 ℓ 。
- **union(A, B)**: 计算并返回集合 $A \leftarrow A \cup B$ 。
- **find(ℓ)**: 返回定位器为 ℓ 的元素所在的集合。

1. 序列实现

总共有 n 个元素的划分的简单实现具有一个序列集合，并且一个序列对应一个集合，其中用于集合 A 的序列存储的元素为定位器结点。每个定位器结点具有一个指向其元素 e 的引用（它允许在 $O(1)$ 时间内执行定位器 ADT 的方法 **element()**）以及一个指向存储 e 的序列的引用。如图 4-5 所示。因此，我们能够在 $O(1)$ 时间内执行操作 **find(ℓ)**。同样，**makeSet** 也需 $O(1)$ 时间。操作 **union(A, B)** 要求将两个序列连接成一个序列，并更新其中一个定位器中的序列引用。我们实现这个操作时，选择将较小序列中的所有定位器删除，并将它们插入到较大的序列中。因此，操作 **union(A, B)** 的运行时间是 $O(\min(|A|, |B|))$ ，即为 $O(n)$ ，因为在最坏情况下， $|A| = |B| = n/2$ 。然而，如下所示，平摊分析表明，这个实现比最坏情况下的实现好得多。

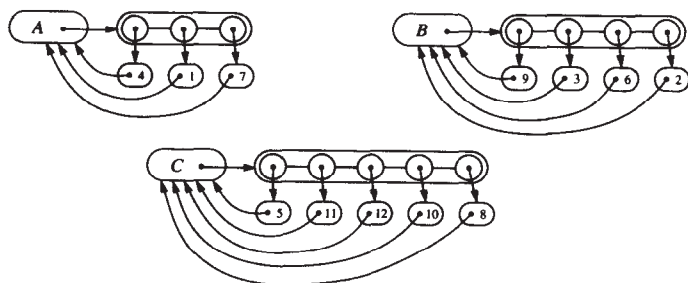


图4-5 包含三个集合的划分基于序列的实现，这三个集合是： $A = \{1, 4, 7\}$ ， $B = \{2, 3, 6, 9\}$ ， $C = \{5, 8, 10, 11, 12\}$

227

2. 序列实现的性能

上述的序列实现是简单的，但也是有效的。正像如下定理表明的那样。

定理4.5 利用上述基于序列的实现，从初始为空的划分开始，执行 n 个makeSet、union和find操作序列的运行时间为 $O(n \log n)$ 。

证明 利用会计方法，假设一个计算机元能够支付执行一次find操作、一次makeSet操作，或者在union操作中将一个定位器结点从一个序列移到另一个序列的时间。在find操作或makeSet操作的情况，为操作自身支付1个计算机元的费用。但是，在union操作的情况，我们向从一个集合移到另一个集合的每个定位器支付1个计算机元的费用。注意，对于union自身并不支付费用。显然，find和makeSet操作的总费用为 $O(n)$ 。

考虑代表union操作支付给定位器的费用。重要的观察是，每次定位器从一个集合移到另一个集合时，新集合的大小至少加倍。因此，每个定位器至多从一个集合移到另一个集合 $\log n$ 次；因此每个定位器至多可以被收取 $O(\log n)$ 次费用。因为我们假设划分初始为空，在给定的操作序列中，引用 $O(n)$ 个不同的元素。这蕴涵着所有union操作的总时间为 $O(n \log n)$ 。 ■

在makeSet、union和find操作序列中，一个操作的平摊运行时间为操作序列的总时间除以操作个数。由上述定理可知，对于利用序列实现的划分，每个操作的平摊运行时间为 $O(\log n)$ 。因此，对于简单的、基于序列的划分实现的性能概括如下：

定理4.6 在 n 个makeSet、union和find组成的操作序列中，利用基于序列的划分实现，从初始为空的划分开始，每个操作的平摊运行时间为 $O(\log n)$ 。

注意在这个基于序列的划分实现中，实际上每个find操作最坏情况下的运行时间为 $O(1)$ 。正是union操作的运行时间成为计算的瓶颈。

在下一节里，我们描述一种基于树的划分实现，它不保证常数时间的find操作，但是每个union操作的平摊运行时间比 $O(\log n)$ 好得多。

228

4.2.3 基于树的划分实现

针对具有 n 个元素的划分的另一种数据结构利用树的集合存储集合中的元素，其中每棵树关联一个不同的集合。如图4-6所示。特别地，我们可以用链表数据结构实现树 T ，其中 T 的每个结点 u 存储关联 T 的集合中的一个元素，parent引用指向 u 的父结点。如果 u 是根，那么它的parent引用指向它自身。树中的结点作为划分中的元素的定位器。同时，我们用它所关联的根标识每个集合。

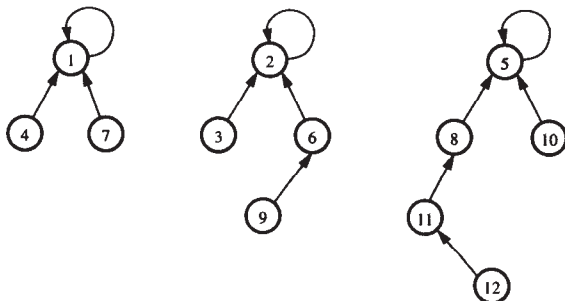


图4-6 基于树的三个不相交集的划分实现： $A = \{1, 4, 7\}$ ， $B = \{2, 3, 6, 9\}$ ， $C = \{5, 8, 10, 11, 12\}$

基于这个划分数据结构，操作union执行将其中的一棵树变成另一棵树的子树的操作，如图4-7b所示。通过修改一棵树根的parent引用，使其指向另一棵树的根来完成这个操作，所需时间为 $O(1)$ 。定位器结点 ℓ 上的操作find可通过向上遍历到包含该定位器的树的根来完成，如图4-7a所示。这个操作最坏情况下的运行时间为 $O(n)$ 。

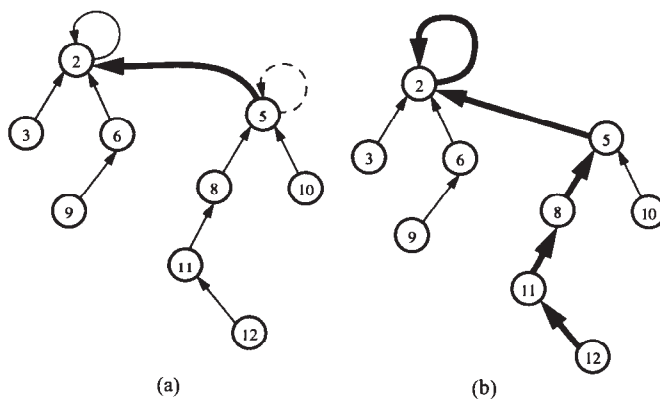


图4-7 基于树的划分实现：(a)操作union(A, B)；(b)操作find(ℓ)，其中 ℓ 表示元素12的定位器

229

注意，树的这种表示是一种特殊的数据结构，用于实现一个划分，它并不表示树抽象数据类型(2.3节)的实现。这种表示方法只有“向上”的链接，并不提供访问给定结点的子结点的方法。初看上去，这种实现并不比基于序列的数据结构实现要好。但是，我们添加如下简单的启发式搜索，使其运行得更快。

按大小合并 (union-by-size)：在每个结点 v 上存储以 v 为根的子树的大小，用 $n(v)$ 表示。在union操作中，使较小集合对应的树成为另一棵树的子树，并更新得到的树的根的大小域。

路径压缩 (path compression)：在find操作中，对于find访问的每个结点 v ，重置 v 的父指针，使其指向根，如图4-8所示。

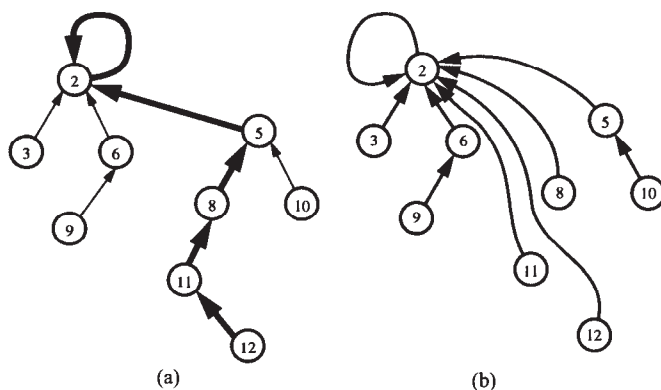


图4-8 路径压缩启发式方法：(a)操作find对元素12的路径遍历；(b)重构树

这些启发式方法使一个操作的运行时间增加了常数因子，但是正如以下所示的那样，它们大大地改进了平摊时间。

1. 定义一个位序函数

在最初包含 n 个单元元素集合的划分上，分析 n 个union和find操作序列的运行时间。

对于每个为根的结点 v ，前面定义 $n(v)$ 为以 v 为根的子树的大小（包括 v ），用集合所关联的树的根表示这个集合。

230

每当把一个集合并入 v 中时，更新 v 的大小域。因此，如果 v 不是根，那么 $n(v)$ 是以 v 为根的可能最大的子树，它恰好出现在将 v 与其他一些结点合并之前，这些结点的大小至少和 v 的大小一样大。对于任意结点 v ，定义 v 的位序（rank） $r(v)$ 为

$$r(v) = \lfloor \log n(v) \rfloor$$

因此，立即可得 $n(v) \geq 2^{r(v)}$ 。同时，由于 v 的树中至多有 n 个结点，对于每个结点 v ，有 $r(v) \leq \lfloor \log n \rfloor$ 。

定理4.7 如果结点 w 是结点 v 的父结点，那么，

$$r(v) < r(w)$$

证明 仅当合并前 w 的大小至少和 v 一样大，才使 v 指向 w 。设 $n(w)$ 表示合并前 w 的大小， $n'(w)$ 表示合并后 w 的大小。因此，合并后可得

$$\begin{aligned} r(v) &= \lfloor \log n(v) \rfloor \\ &< \lfloor \log n(w) + 1 \rfloor \\ &= \lfloor \log 2n(w) \rfloor \\ &\leq \lfloor \log (n(v) + n(w)) \rfloor \\ &= \lfloor \log n'(w) \rfloor \\ &\leq r(w) \end{aligned}$$

换句话说，这个定理说明，当在树中沿着parent指针向上行进时，位序单调（monotonically）递增。同时，它还蕴涵如下定理：

定理4.8 对于 $0 \leq s \leq \lfloor \log n \rfloor$ ，至多存在 $n/2^s$ 个位序为 s 的结点。

证明 由定理4.7可知，对于父结点为 w 的任何结点 v ， $r(v) < r(w)$ 。当在树中沿着parent指针向上行进时，位序单调递增。因此，对于结点 v 和 w ，如果 $r(v) = r(w)$ ，那么 $n(v)$ 中统计的结点数一定独立于并且不同于 $n(w)$ 中统计的结点数。由位序的定义可知，如果一个结点 v 的位序为 s ，那么 $n(v) \geq 2^s$ 。于是，由于至多共有 n 个结点，因而也至多可能有 $n/2^s$ 个位序为 s 的结点。

2. 平摊分析

当用按大小合并及路径压缩的启发式搜索实现基于树的划分数据结构时，它有一个令人惊奇的性质，即进行 n 个union和find操作的序列所花费的时间为 $O(n \log^* n)$ ，其中 $\log^* n$ 是“迭代对数”函数，它是2的幂（tower-of-twos）函数 $t(i)$ 的逆：

$$t(i) = \begin{cases} 1 & \text{如果 } i = 0 \\ 2^{t(i-1)} & \text{如果 } i \geq 1 \end{cases}$$

即

231

$$\log^* n = \min \{i: t(i) \geq n\}$$

直觉上， $\log^* n$ 是对某个数不断取以2为底的对数（在得到小于2的数之前）所重复的次数。表4-1显示了这个函数的几个值。

正如表4-1所示，对于实际有意义的值， $\log^* n \leq 5$ 。这是一个增长速度慢得出奇的函数（但是却在增长）。

表4-1 $\log^* n$ 的某些样本值, 以及获得这个值所需的最小值 n

最小值 n	$\log^* n$
1	0
2	1
$2^2 = 4$	2
$2^{2^2} = 16$	3
$2^{2^{2^2}} = 65\,536$	4
$2^{2^{2^{2^2}}} = 2^{65\,536}$	5

为了证实我们的令人吃惊的声明, 执行 n 个union和find操作所需的时间为 $O(n \log^* n)$, 我们把结点划分成位序组 (rank group)。如果

$$g = \log^*(r(v)) = \log^*(r(u))$$

则称结点 v 和 u 在同一位序组 g 中。

由于最大位序为 $\lfloor \log n \rfloor$, 则最大位序组为

$$\log^*(\log n) = \log^* n - 1$$

我们通过会计方法利用位序组导出平摊分析的规则。执行一次union所需时间为 $O(1)$ 。并为每个union操作支付1个计算机元。因此, 我们可以把注意力集中在find操作上, 证实我们的声明, 执行 n 个union和find操作的时间为 $O(n \log^* n)$ 。

执行find操作的主要计算任务是, 在树中沿着parent指针从某个结点 u 上行至包含 u 的树根。对于遍历的每个parent引用支付一个计算机元。设 v 是此路径上的某个结点, w 是 v 的父结点。利用两个规则, 支付遍历这个parent引用的费用:

- 如果 w 是根, 或者如果 w 与 v 不在同一位序组中, 那么支付find操作一个计算机元。
- 否则 (w 不是根, 且 v 和 w 在同一个位序组中), 支付结点 v 一个计算机元。

因为至多有 $\log^* n - 1$ 个位序组, 这个规则保证任何find操作至多被支付 $\log^* n$ 个计算机元。因此, 这种模式计算了find操作的费用, 但我们还必须计算所有支付给结点的费用。

232

观察可见, 在支付了结点 v 的费用之后, v 得到一个新的父结点, 它是 v 的树中更高一层上的结点。此外, 由于位序单调递增, v 的新父结点的位序会大于 v 的原父结点 w 的位序。因此, 任何结点 v 最多可以被支付 v 的位序组中不同位序数量的费用。同时, 由于位序组0中结点的父结点在更高的位序组中, 可以只考虑比0大的位序组中的结点 (因为总是支付find操作, 检查位序组0中的结点)。如果 v 在 $g > 0$ 的位序组中, 那么在 v 有一个在更高位序组中的父结点之前, 至多会被支付 $t(g) - t(g-1)$ 次 (此后, v 不再被支付费用)。换句话说, 支付给结点的总费用 (计算机元) C 可被限定为

$$C \leq \sum_{g=1}^{\log^* n - 1} n(g) \cdot (t(g) - t(g-1))$$

其中 $n(g)$ 表示位序组 g 中的结点数。

于是, 如果导出 $n(g)$ 上界, 就可代入上式, 得到 C 的上界。为了导出位序组 g 中结点数上界 $n(g)$, 回忆任一给定位序为 s 的组中结点数至多为 $n/2^s$ (由定理4.8可知)。因此, 对于 $g > 0$,

$$\begin{aligned}
n(g) &\leq \sum_{s=t(g-1)+1}^{t(g)} \frac{n}{2^s} \\
&= \frac{n}{2^{t(g-1)+1}} \sum_{s=0}^{t(g)-t(g-1)-1} \frac{1}{2^s} \\
&< \frac{n}{2^{t(g-1)+1}} \cdot 2 \\
&= \frac{n}{2^{t(g-1)}} \\
&= \frac{n}{t(g)}
\end{aligned}$$

将此界限代入结点总费用 C 中, 得

$$\begin{aligned}
C &< \sum_{g=1}^{\log^* n - 1} \frac{n}{t(g)} \cdot (t(g) - t(g-1)) \\
&\leq \sum_{g=1}^{\log^* n - 1} \frac{n}{t(g)} \cdot t(g) \\
&= \sum_{g=1}^{\log^* n - 1} n \\
&\leq n \log^* n
\end{aligned}$$

233

于是, 证明了支付给结点的计算机元至多为 $n \log^* n$ 。这蕴涵如下定理:

定理4.9 设 P 是树集合实现的具有 n 个元素的划分, 树集合如上所述, 该实现利用了按大小合并及路径压缩的启发式方法。在 P 上执行一系列union和find操作, 从单元素集的集合开始, 每个操作的平摊运行时间为 $O(\log^* n)$ 。

3. Ackermann函数

可以证明, 在上述实现的 n 个划分操作中, 一个操作的平摊运行时间为 $O(\alpha(n))$, 其中 $\alpha(n)$ 是一个函数, 称为Ackermann逆函数 A , 它是一个渐近增长函数, 其增长速度甚至比 $\log^* n$ 还慢, 但是其证明超出了本书的范围。

这里仍要定义Ackermann函数, 以便详细分析它快速增长及其逆函数慢速增长的程度。 A_i 定义如下:

$$\begin{aligned}
A_0(n) &= 2n && \text{对于 } n \geq 0 \\
A_i(1) &= A_{i-1}(2) && \text{对于 } i \geq 1 \\
A_i(n) &= A_{i-1}(A_i(n-1)) && \text{对于 } i \geq 1 \text{ 且 } n \geq 2
\end{aligned}$$

换句话说, 带下标的Ackermann函数定义了一个函数级数, 并且每个函数比前一个函数的增长速度快得多:

- $A_0(n) = 2n$ 是2的倍数函数。
- $A_1(n) = 2^n$ 是2的幂函数。
- $A_2(n) = 2^{2^{\cdot^{\cdot^{\cdot}}}}$ 是2的塔函数。
- $A_3(n)$ 是2的塔的塔函数。
- 依此类推。

定义Ackermann函数为 $A(n) = A_n(n)$, 这是一个其增长速度快得难以置信的函数。同样, 它的

逆函数 $\alpha(n) = \min\{m: A(m) \geq n\}$ 是一个其增长速度慢得难以置信的函数。

回到排序问题，讨论快速排序。像归并排序一样，这个算法也基于分治法范型，但是以在一定程度上相反的方式应用这项技术，因为所有困难的工作都会在递归调用之前完成。

234

4.3 快速排序

快速排序算法利用简单分治法对序列 S 排序，将 S 分解成子序列，递归对每一子序列进行排序，然后通过简单的连接把有序的子序列组合起来。快速排序算法包含以下三个步骤（如图4-9所示）：

(1) 划分：如果 S 中至少有两个元素（如果 S 中有0个或1个元素，则无需执行任何动作），从 S 中选择某一元素 x ，称这个元素为枢轴元素（pivot）。通常，选择 S 中最右边的元素。删除 S 中的所有元素，并将它们放入三个序列中：

- L ，存储 S 中小于 x 的元素。
- E ，存储 S 中等于 x 的元素。
- G ，存储 S 中大于 x 的元素。

（如果 S 中的元素各不相同，则 E 只存储一个元素，即枢轴元素）。

(2) 递归：递归对 L 和 G 进行排序。

(3) 求解：把元素放回 S 中，其顺序是：首先插入 L 中的元素，然后插入 E 中的元素，最后插入 G 中的元素。

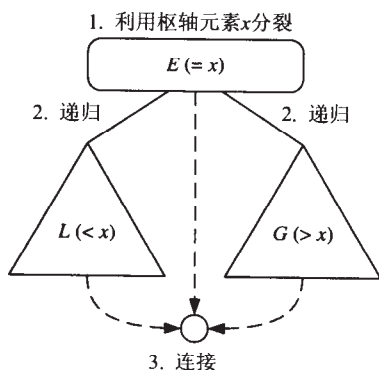


图4-9 快速排序算法的可视化示意图

像归并排序一样，可以利用二叉递归树可视化快速排序的过程，称该树为快速排序树。图4-10显示了快速排序算法的可视化过程，给出了快速排序树中每个结点上的示范输入和输出序列。

不像快速排序，关联快速排序一次执行的快速排序树的高度最坏情况下是线性的。例如，如果对由 n 个不同元素组成的有序序列进行排序，就会出现这种情况。在这种情况下，按照选择枢轴元素的标准，选择最大元素作为枢轴元素，产生大小为 $n-1$ 的子序列 L ，大小为1的子序列 E ，以及大小为0的子序列 G 。因此，快速排序树的高度在最坏情况下为 $n-1$ 。

235

快速排序的运行时间

利用4.1.1节分析归并排序使用的技术，分析快速排序的运行时间。也就是说，首先表示快速排序树 T 中每个结点所花费的时间（图4-10），然后把所有结点上的运行时间相加。快速排序的划分步骤和求解步骤用线性时间即可实现。因此， T 中的结点 v 上所花费的时间与其输入大小 $s(v)$ 成正比，定义为调用关联结点 v 的快速排序时所处理的序列大小。由于子序列 E 中至少有一个元素（枢轴元素）， v 的子结点的输入大小之和至多为 $s(v)-1$ 。

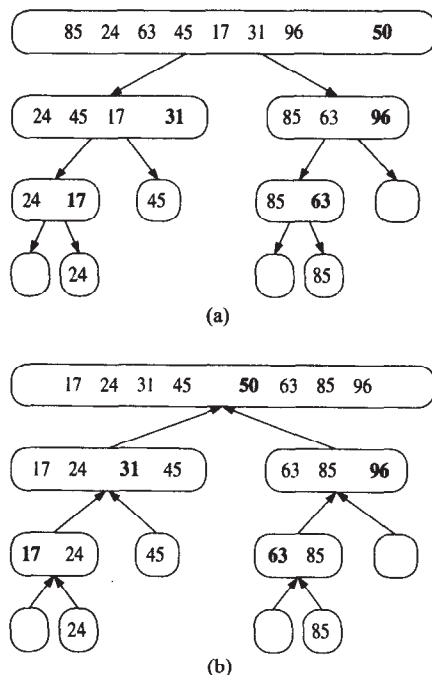


图4-10 用于在8个元素的序列上执行快速排序算法的快速排序树 T : (a)在 T 中每个结点上处理的输入序列; (b)在 T 中每个结点上产生的输出序列。每层递归所用的枢轴元素用粗体显示

给定一棵快速排序树 T , 设 s_i 表示 T 中深度为 i 的结点的输入大小之和。显然, $s_0 = n$, 因为 T 的根 r 关联整个序列。同样, $s_1 \leq n-1$, 这是由于枢轴元素并不会传播到 r 的子结点中。考虑下一个 s_2 , 如果 r 的两个子结点的输入大小不为 0, 那么, $s_2 = n-3$; 否则 (根的其中一个子结点的大小为 0, 另一个的大小为 $n-1$), $s_2 = n-2$ 。因此, $s_2 \leq n-2$ 。继续这个推理过程, 可得 $s_i \leq n-i$ 。

236

正如在 4.3 节中看到的那样, 最坏情况下, T 的高度为 $n-1$ 。因此, 快速排序最坏情况下的运行时间是

$$O\left(\sum_{i=0}^{n-1} s_i\right), \text{ 为 } O\left(\sum_{i=0}^{n-1} (n-i)\right), \text{ 即 } O\left(\sum_{i=1}^n i\right)$$

由定理 1.3 可知, $\sum_{i=1}^n i$ 为 $O(n^2)$ 。因此, 快速排序最坏情况下的运行时间为 $O(n^2)$ 。

据其名称, 我们希望快速排序快速运行。然而, 上述二次时间界限表明快速排序最坏情况下运行较慢。然而, 矛盾的是, 这种最坏情况下的行为出现在输入序列是有序的情况下。注意, 如果对不同元素组成的序列进行排序的结果出现最好情况, 此时子序列 L 和 G 的大小大约相等。实际上, 在这种情况下, 每个内部结点存储一个枢轴元素, 并调用其大小相等的两个子结点。因此, 在根处存储 1 个枢轴元素, 在第 1 层存储 2 个枢轴元素, 在第 2 层存储 2^2 个枢轴元素, 依此类推。也就是在最好情况下, 有

$$\begin{aligned} s_0 &= n \\ s_1 &= n-1 \\ s_2 &= n - (1+2) = n-3 \\ &\vdots \\ s_i &= n - (1 + 2 + 2^2 + \cdots + 2^{i-1}) = n - (2^i - 1) \end{aligned}$$

依此类推。因此，在最好情况下， T 的高度为 $O(\log n)$ ，快速排序的运行时间为 $O(n \log n)$ 。这个事实的证明过程留作习题（习题R-4.11）。

直觉上，快速排序非形式的期望行为是，每次调用时枢轴元素把输入序列分成大致相等的部分。因此，期望快速排序的运行时间类似于最好情况下的运行时间 $O(n \log n)$ 。下一节里，通过引入随机化过程，使得快速排序的行为就像上面所描述的那样。

随机化快速排序

分析快速排序的一种普遍的方法是，假设枢轴元素总是把序列分成几乎相等的部分。这样的假设预先假定了通常不可用的输入分布的知识。例如，我们一定会假设输入序列是“几乎”有序序列的情况很少出现，实际上，在许多应用中，常见到这种情况。幸运的是，不需要这个假设，以将我们的直觉与快速排序的行为联系起来。

因为快速排序划分步骤的目标是把序列 S 分成几乎相等的部分。我们利用一种新的规则选择枢轴元素——选择输入序列中的随机元素（random element）作为枢轴元素。如下所见，称所得算法为随机化快速排序（randomized quick-sort）。对于给定的由 n 个元素组成的序列，期望运行时间为 $O(n \log n)$ 。

237

定理4.10 对于大小为 n 的序列，随机化快速排序期望运行时间为 $O(n \log n)$ 。

证明 利用以下概率论中的事实：

不断投掷一枚均匀的硬币，直到显示 k 次正面的期望投掷次数为 $2k$ 。

考虑随机化快速排序的某次递归调用，设 m 表示本次调用的输入序列的大小。如果选择的枢轴元素产生子序列 L 和 G 的大小至少为 $m/4$ ，至多为 $3m/4$ ，则称这次调用是“良好的”。由于随机均匀选择枢轴元素，则有 $m/2$ 个枢轴元素会产生良好的调用。出现良好调用的概率为 $1/2$ （同硬币出现正面的概率一样）。

如果快速排序树 T 中的一个结点 v 关联一次良好的调用（如图4-11所示），那么， v 的每个子结点的输入大小至多为 $3s(v)/4$ （等于 $s(v)/(4/3)$ ）。如果取 T 中从根到外部结点的任一路径，那么这条路径长度至多为达到 $\log_{4/3} n$ 次良好调用所进行的调用次数（这条路径上的每个结点）。应用上述概率知识，可知出现 $\log_{4/3} n$ 次良好的调用，期望必须进行调用的次数为 $2\log_{4/3} n$ （如果路径在此层之前终止，结果会更好）。因此，从根到 T 中外部结点的任一期望路径长度为 $O(\log n)$ 。回忆可知， T 中每一层所花的时间为 $O(n)$ ，因而随机化快速排序的期望运行时间为 $O(n \log n)$ 。■

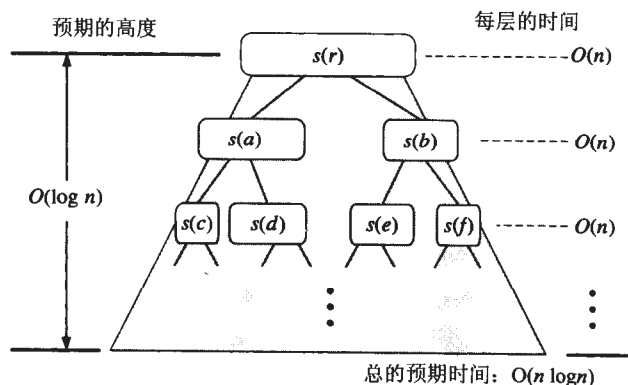


图4-11 快速排序树 T 的可视化时间分析

注意，期望的运行时间是取算法可能做出的所有选择，它与算法可能遇见的输入序列的分布

238

无关。实际上，利用概率论事实，可以证明随机化快速排序的运行时间为 $O(n \log n)$ 的概率较高（见习题C-4.8）。

4.4 基于比较的排序下界

到目前为止所讨论的排序算法描述了序列上的几个最坏情况或期望运行时间为 $O(n \log n)$ 的方法，其中输入序列的大小为 n 。这些方法包括本章中描述的归并排序和快速排序，以及2.4.4节描述的堆排序。一个自然会问到的问题是，是否可能存在比 $O(n \log n)$ 时间更快的排序算法。

在这一节里讨论如果一个排序算法所用的基本操作是比较两个元素，那么这是我们所能得到的最好结果。基于比较的排序最坏情况下运行时间的下界为 $\Omega(n \log n)$ （回忆1.2.2节关于符号 $\Omega(\cdot)$ 的定义）。为了强调基于比较排序算法的主要开销，我们只计算一个排序算法所执行的比较次数。因为这足以导出一个算法的下界。

假定已知待排序序列 $S = (x_0, x_1, \dots, x_{n-1})$ ，假设所有 S 中的元素互不相同（这不是限制条件，因为我们导出的是下界）。每当一个排序算法比较两个元素 x_i 和 x_j （即 $x_i < x_j?$ ）时，则有两种可能的结果：“yes”或“no”。基于比较的结果，排序算法可能执行某些内部计算（这里没有考虑），最后执行 S 中其他两个元素的比较，又会产生两个结果。于是，可用一棵判定树 T 表示基于比较的排序算法。也就是说， T 中的每个内部结点对应一次比较，从结点 v 到其子结点的边对应“yes”或“no”结果的计算。如图4-12所示。

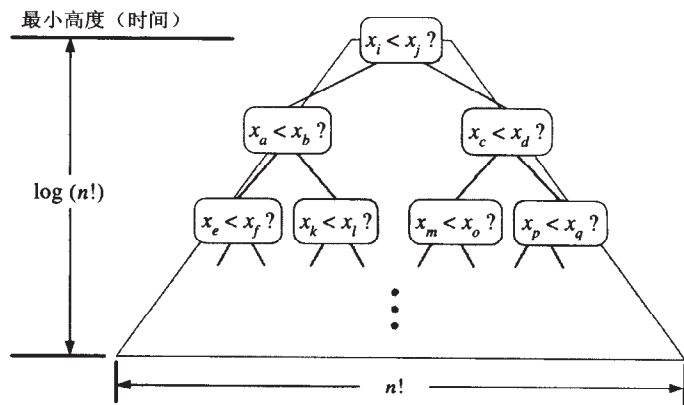


图4-12 可视化基于比较排序的下界

重要的是注意到，假定所讨论的排序算法可能没有明确的关于树 T 的知识。我们隐含着利用 T 表示一个排序算法所有可能的比较序列，从第一次比较开始（关联根），结束于算法执行终止之前的最后一次比较（关联一个外部结点的父结点）。

S 中元素的每种可能的初始次序或排列（permutation）都会导致排序算法执行一系列比较，从而遍历一条从根到某些外部结点的路径。 T 中每个外部结点 v 关联 S 的一个排列集，使得排序算法结束于 v 。在下界证明中最重要的观察是， T 中的每个外部结点可以表示 S 的至多一个排列的比较序列。证明的过程相当简单：如果 S 的两个不同排列 P_1 和 P_2 关联同一个外部结点，那么，至少存在两个对象 x_i 和 x_j ，满足在 P_1 中 x_i 位于 x_j 之前，但是在 P_2 中 x_i 位于 x_j 之后。同时，关联 v 的输出一定是 S 的某种重新排序，并且 x_i 或 x_j 出现在另一个对象之前。但是，如果 P_1 和 P_2 都导致排序算法按同样次序输出 S 中的元素，那么，蕴涵着存在一种方式使算法按照错误的次序输出 x_i 和 x_j 。因为这是

239

正确排序算法所不允许的,因此 T 中的每个外部结点一定正好只关联 S 的一个排列。利用与排序算法关联的判定树的这个性质证明以下结论:

定理4.11 基于比较对 n 个元素的序列进行排序的算法,最坏情况下的运行时间为 $\Omega(n \log n)$ 。

证明 正如上述描述的那样,基于比较的排序算法的运行时间一定大于或等于关联此算法的判定树 T 的高度。如图4-12所示。由上述证明, T 中的每个外部结点一定关联 S 的一个排列。然而, S 的每个排列必定会导致 T 中一个不同的外部结点。 n 个对象的排列数为

$$n! = n(n-1)(n-2) \cdots 2 \cdot 1$$

因此, T 至少有 $n!$ 个外部结点。由定理2.2可知, T 的高度至少为 $\log(n!)$ 。这直接证明了这个定理,因为在乘积 $n!$ 中,至少有 $n/2$ 个项大于或等于 $n/2$;因此,

$$\log(n!) \geq \log(n/2)^{n/2} = (n/2) \log(n/2)$$

即为 $\Omega(n \log n)$ 。

240

4.5 桶排序和基数排序

在前一节里,证明了对于在最坏情况下对 n 个元素的序列进行排序的比较算法, $\Omega(n \log n)$ 时间是一个必要条件。一个自然要询问的问题是,是否存在其他类型的排序算法,其渐近运行时间比 $O(n \log n)$ 要快。令人感兴趣的是,这样的算法存在,但是这些算法要求对需排序的输入序列做出特定假设。即便如此,这样的情况常常出现在实际中,因此值得讨论它们。在这一节里,考虑对数据项组成的序列进行排序的问题,其中每个数据项是一个关键字-元素对。

4.5.1 桶排序

考虑 n 个数据项的序列 S ,其关键字是 $[0, N-1]$ 区间内的整数,对于某个整数 $N \geq 2$,假定要按照数据项的关键字排序。在这种情况下,可用 $O(n + N)$ 时间对 S 排序。看起来结果是令人惊讶的,但这蕴涵着,例如如果 $N = O(n)$,则可用 $O(n)$ 时间对 S 排序。当然,问题的关键是:由于对元素的格式做了限制性假设,我们可以避免使用比较。

主要思想是利用称为桶排序(bucket-sort)的算法。这不是基于比较的算法,而是基于把关键字作为桶数组 B 的下标,其范围为 $0 \sim N-1$ 。关键字为 k 的数据项放在“桶” $B[k]$ 中, $B[k]$ 自身是一个数据项(关键字为 k)的序列,将输入序列 S 中的每个数据项插入桶中之后,通过按顺序枚举桶 $B[0], B[1], \dots, B[N-1]$ 中的内容,把数据项按顺序放回 S 中。算法4-2给出了桶排序的伪代码描述。

算法4-2 桶排序

算法 bucketSort(S):

输入: 关键字位于 $[0, N-1]$ 区间内的数据项组成的序列 S

输出: 按照关键字非降序排列的有序序列 S

 设 B 是 N 个序列的数组,每个序列初始为空

for S 中的每个数据项 x **do**

 设 k 是 x 的关键字

 从 S 中删除 x ,并将它插入到桶(序列) $B[k]$ 的末尾

for $i \leftarrow 0$ **to** $N-1$ **do**

for 序列 $B[i]$ 中的每个数据项 x **do**

 从 $B[i]$ 中删除 x ,并将它插入到 S 的末尾

[241]

由上述算法可知,桶排序算法的运行时间为 $O(n + N)$,占用空间为 $O(n + N)$,这只需要考察算法中的两个for循环。

因此,与输入大小 n 相比,当关键字的数值范围 N 较小时,桶排序是有效算法,即 $N = O(n)$ 或 $N = O(n \log n)$ 。但是,与 n 相比,当 N 不断增长时,算法的性能越来越差。

此外,桶排序算法的一个重要性质是,即使有许多不同元素具有相同的关键字,算法仍然是正确的。实际上,我们对它的描述预期满足这种情况。

稳定排序

当对关键字-元素对的数据项进行排序时,一个重要的问题是如何处理相同的关键字。设 $S = ((k_0, e_0), \dots, (k_{n-1}, e_{n-1}))$ 是由数据项组成的序列。如果对于 S 的任意两个数据项 (k_i, e_i) 和 (k_j, e_j) ,满足 $k_i = k_j$,且排序之前数据项 (k_i, e_i) 在 (k_j, e_j) 之前,排序之后数据项 (k_i, e_i) 仍在 (k_j, e_j) 之前。则称排序算法是稳定的(stable)。稳定性对于一个排序算法是重要的,因为某些应用希望保持关键字相同的那些元素的初始次序。

算法4-2中给出的桶排序算法的非形式描述并不能保证算法的稳定性。这种稳定性不是桶排序算法自身中固有的,然而,可以轻松修改我们的描述,使得它是稳定的算法,同时保持它的运行时间 $O(n + N)$ 不变。实际上,在算法执行过程中,总是删除序列 S 和序列 $B[i]$ 中的第一个(first)元素,可得稳定的桶排序算法。

4.5.2 基数排序

稳定排序之所以如此重要的原因之一是,桶排序算法可用于更广泛的环境中,而不仅限于对整数进行排序。假定要对关键字为数对 (k, l) 的数据项进行排序,其中 k 和 l 是 $[0, N-1]$ 区间内的整数, $N \geq 2$ 为整数。在这样一个环境中,自然会利用字典次序(lexicographical)约定来定义这些数据项的次序,如果

- $k_1 < k_2$ 或
- $k_1 = k_2$ 且 $l_1 < l_2$

则 $(k_1, l_1) < (k_2, l_2)$ 。

这是字典比较函数的点对形式,通常可用于等长字符串(容易推广到 d 个数的元组上,其中 $d > 2$)。

基数排序(radix sort)算法对序列 S 应用两次稳定桶排序算法,完成对数对序列 S 的排序;首先利用数对中的一个分量作为排序关键字,然后利用第二个分量。但哪个次序是正确的?首先应该对 k 分量(第一个分量)排序,然后,再对 l 分量(第二个分量)排序,或者应该还有其他方式吗?

[242]

在回答这个问题之前,先考虑如下例子。

示例4.2 考虑如下序列 S :

$$S = ((3, 3), (1, 5), (2, 5), (1, 2), (2, 3), (1, 7), (3, 2), (2, 2))$$

如果对 S 的第一个分量进行稳定排序,则可得如下序列

$$S_1 = ((1, 5), (1, 2), (1, 7), (2, 5), (2, 3), (2, 2), (3, 3), (3, 2))$$

然后对 S_1 的第二个分量进行稳定排序,则可得如下序列

$$S_{1,2} = ((1, 2), (2, 2), (3, 2), (2, 3), (3, 3), (1, 5), (2, 5), (1, 7))$$

这不完全是一个有序序列。另一方面,如果首先对 S 的第二个分量进行稳定排序,则得如下序列

$$S_2 = ((1, 2), (3, 2), (2, 2), (3, 3), (2, 3), (1, 5), (2, 5), (1, 7))$$

然后对序列 S_2 的第一个分量进行稳定排序, 则可得如下序列

$$S_{2,1} = ((1, 2), (1, 5), (1, 7), (2, 2), (2, 3), (2, 5), (3, 2), (3, 3))$$

它确实是一个字典有序序列 S 。

由此例可知, 应该首先利用第二个分量排序, 然后利用第一个分量排序。这种直觉完全正确。首先利用第二个分量进行稳定排序, 然后再利用第一个分量, 可以保证: 如果在第二个排序(利用第一个分量)中, 存在两个相等的元素, 则它们在初始序列中的相对顺序保持不变。因此, 这样可以保证每次得到的序列是有序的。如何把这种方法扩展到三元组或是其他 d 元组留作习题(R-4.15)。概括如下:

定理4.12 设 S 是 n 个关键字-元素的数据项组成的序列, 每个数据项的关键字为 (k_1, k_2, \dots, k_d) , 其中 k_i 是 $[0, N-1]$ 区间内的整数, $N \geq 2$ 。利用基数排序把 S 排成字典次序的时间为 $O(d(n + N))$ 。

排序不仅仅是处理元素集上全序关系的一个有趣问题, 还有许多其他应用。例如, 在这些应用中, 不需要整个集合上的一个有序表, 而只要求集合中的部分信息的一个顺序关系。在研究这样一个问题(称为“选择”)之前, 先回顾目前学过的排序算法, 并做简要比较。

243

4.6 比较排序算法

现在, 我们放松一下, 考虑本书中讨论的针对 n 个元素的序列的所有排序算法。对于生活中的许多事物, 并不存在真正的“最佳”排序算法。但是这里针对“良好”算法的某些已知性质, 可以给出一些指导和观察。

如果一个算法实现得很好, 例如, 插入排序的运行时间为 $O(n + k)$, 其中 k 为逆序数(即呈反序的元素对个数)。因此, 对于较小的序列(如少于50个元素), 插入排序是一种优秀的排序算法。同时, 对于“几乎”有序的序列, 插入排序也是相当有效的。“几乎”的含义是, 序列中的逆序数非常小。但是, 除了这些特殊情况之外, $O(n^2)$ 的运行时间使得插入排序算法的性能相当不佳。

另一方面, 归并排序最坏情况下的运行时间为 $O(n \log n)$ 。对于基于比较的排序算法而言, 它是最优算法。然而, 实验研究表明, 难以对归并排序进行原位排序。因此, 从计算机主存区域有限的角度考虑, 与可以进行原位排序的堆排序和快速排序相比, 归并排序不具吸引力。即便如此, 对于那些输入不能完全装入主存中, 而必须按块存放在外部存储器(如磁盘)中的应用问题, 归并排序仍然是一个优秀的算法。在这些环境中, 归并排序过程运行较长归并流中的数据, 使它可以充分利用以块形式引入主存的数据(见14.1.3节)。

实验研究表明, 如果输入序列能够完全装入主存, 那么原位快速排序和堆排序比归并排序运行时间短。事实上, 平均而言, 在测试中快速排序是这些排序算法中最好的。因而, 作为通用的内存排序算法, 快速排序是最佳选择。实际上, 它包含在C语言的库提供的qsort实用程序中。在实时应用中, 要保证一个排序操作的时间, 由于快速排序最坏情况下具有 $O(n^2)$ 的时间性能, 所以它不是一个好的选择。

在实时情况下, 执行排序操作的时间是固定的, 输入数据可装入内存, 堆排序也是最佳选择。它的最坏情况下的运行时间为 $O(n \log n)$, 并且易于执行原位排序。

最后, 如果应用涉及对整型关键字或者整型关键字的 d 元组排序, 那么桶排序或基数排序是最佳选择。因为它的运行时间为 $O(d(n + N))$, 其中 $[0, N-1]$ 是整型关键字的($d = 1$ 即为桶排序)

所属区间。因此，如果 $d(n+N)$ 比 $O(n \log n)$ 小（形式上， $d(n+N) = o(n \log n)$ ），那么，这种排序方法甚至比快速排序或堆排序运行得更快。

244 因此，对不同排序算法的研究为算法设计“工具箱”提供了通用的排序算法集合。

4.7 选择

有许多应用，我们只对其中某一个元素在整个集合中的位序感兴趣。例子包括找出最小元素和最大元素，还可能对（比如）确定中值（median）元素感兴趣，即满足有一半元素比它小，其余的元素比它大。一般而言，查询具有给定位序的某个元素称为顺序统计（order statistics）。

在这一节里，讨论从 n 个可比较的无序元素集中选择第 k 个最小元素的顺序统计问题。称之为选择（selection）问题，当然，可以通过对集合进行排序，并取位序下标为 $k-1$ 的元素来求解该问题。利用最好的基于比较的排序算法，这种方法所需时间为 $O(n \log n)$ 。因此，一个自然要问到的问题是，对于所有 k 值，是否能够用 $O(n)$ 时间找出第 k 个最小元素？包括找出中值元素 $k = \lceil n/2 \rceil$ 。

4.7.1 剪枝-查找法

这听起来有些令人惊讶，但对于所有 k 值，我们的确可以用 $O(n)$ 时间求解选择问题。为了得到这个结果所用的技术涉及一种有趣的算法设计方法。这种算法设计方法称为剪枝-查找法（prune-and-search）或减小求解法（decrease-and-conquer）。在应用这种设计模式时，给定的问题定义在 n 个对象的集合上，通过剪掉 n 个对象中一部分，递归地求解更小的子问题，最终使问题大小减小到一个定义的对象集的常量大小上。然后，利用某种蛮力方法直接求解这个大小为常量的问题，并从所有递归调用中返回，完成构造过程。在某些情况下，可以不用递归，简单反复使用剪枝-查找归约步骤，直到可以利用蛮力方法并终止。

4.7.2 随机化快速选择

在应用剪枝-查找模式求解选择问题时，可以设计一种简单、实用的方法，称为随机化快速选择法（randomized quick-select），用于在定义了全序关系的 n 个元素的无序序列中选择第 k 个最小的元素。随机化快速选择算法取所有可能的随机情况，期望（expected）运行时间为 $O(n)$ 。这个期望的运行时间与对输入分布所做的随机性假设无关。尽管随机化快速选择算法最坏情况下的运行时间为 $O(n^2)$ ，但其证明过程留作习题（R-4.18）。还给出了一个习题（C-4.24），要求对随机化快速选择算法进行修改，得到最坏情况下 $O(n)$ 时间的确定性（deterministic）选择算法。然而，确定性算法的存在主要出于理论上的兴趣，因为在这种情况下，被大 O 符号隐藏的常数因子相对较大。

假定给定 n 个可比较元素的无序序列 S ，以及一个整数 $k \in [1, n]$ 。在较高层次上，找出 S 中第 k 个最小元素的快速选择算法在结构上类似于4.3.1节中描述的随机化快速排序算法。从 S 中随机选取一个元素 x ，并利用这个元素作为“枢轴”元素，把 S 划分成 L 、 E 和 G 三个子序列。分别用于存放小于 x 的元素，等于 x 的元素和大于 x 的元素。这称为剪枝步骤。然后基于 k 的值，确定对这些集中的哪一个进行递归。算法4-3中描述了随机化快速选择算法。

算法4-3 随机化快速选择算法

```

算法 quickSelect( $S, k$ ):
    输入:  $n$ 个可比较的元素的序列 $S$ ，以及一个整数 $k \in [1, n]$ 
    输出:  $S$ 的第 $k$ 个最小的元素
    if  $n = 1$  then

```

```

    return S的(第一个)元素
    随机选取S中的一个元素x
    从S中删除所有元素, 并将它们放入三个序列中:
        • L, 存放小于x的元素
        • E, 存放等于x的元素
        • G, 存放大于x的元素
    if k ≤ |L| then
        quickSelect(L, k)
    else if k ≤ |L| + |E| then
        return x    {E中的每个元素都等于x}
    else
        quickSelect(G, k - |L| - |E|) {注意新的选择参数}

```

4.7.3 随机化快速选择分析

上述提到的随机化快速选择算法期望运行时间为 $O(n)$ 。幸运的是, 证明这个结论只要求最简单的概率知识。所用的主要概率知识是数学期望的线性性 (linearity of expectation)。回忆这样一个事实, 如果 X 和 Y 是随机变量, c 是一个数, 则有 $E(X + Y) = E(X) + E(Y)$ 且 $E(cX) = cE(X)$, 其中利用 $E(Z)$ 表示表达式 Z 的期望值。

246

设 $t(n)$ 表示对大小为 n 的序列执行随机化快速选择算法的运行时间。因为随机化快速选择算法与随机事件的发生有关, 因而, 它的运行时间 $t(n)$ 也是一个随机变量。我们只对 $E(t(n))$ 的界限即 $t(n)$ 的期望值感兴趣。如果一个随机化快速选择算法对 S 进行划分, 导致 L 和 G 的大小至多为 $3n/4$, 则称它的递归调用是“良好”的。显然, 出现良好递归调用的概率为 $1/2$ 。设 $g(n)$ 表示得到一个良好的递归调用之前进行的连续递归调用 (包括当前调用) 的次数, 那么

$$t(n) \leq bn \cdot g(n) + t(3n/4)$$

其中 $b > 0$ 是常数 (表示每次调用的开销)。当然, 集中考虑 $n > 1$ 的情况, 因为很容易用封闭形式表征 $t(1) = b$ 时的情况。由期望的线性性可知,

$$E(t(n)) \leq E(bn \cdot g(n) + t(3n/4)) = bn \cdot E(g(n)) + E(t(3n/4))$$

因为出现良好递归调用的概率为 $1/2$, 无论递归调用是好是坏, 都与它的父递归调用是好是坏无关, 期望值 $g(n)$ 就是投掷均匀硬币出现“正面”需要投掷的次数。这蕴涵着 $E(g(n)) = 2$ 。因此, 如果设 $T(n)$ 是 $E(t(n))$ 的简写表示符号 (即随机化快速选择算法的期望运行时间), 那么, 对于 $n > 1$, 有

$$T(n) \leq T(3n/4) + 2bn$$

如同归并排序方程, 希望把这个方程转换成封闭形式。为此, 假定 n 较大, 反复应用方程。例如, 在经过两次迭代应用后, 得到

$$T(n) \leq T((3/4)^2 n) + 2b(3/4)n + 2bn$$

此时, 可得一般形式, 如下

$$T(n) \leq 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i$$

换句话说, 随机化快速选择的期望运行时间是几何级数和的 $2bn$ 倍, 该几何级数的底为小于1的正数。因此, 由关于几何级数的定理1.2可知 $T(n)$ 为 $O(n)$ 。概括为如下定理:

定理4.13 对于大小为 n 的序列, 随机化快速选择算法的期望运行时间为 $O(n)$ 。

247

正如早先提到的那样，还有另一个快速选择算法的变体，它没有利用随机化思想，其最坏情况下的运行时间为 $O(n)$ 。习题C-4.24就是为对设计和分析此算法感兴趣的读者设计的。

4.8 Java 示例：原位快速排序

回忆2.4.4节可知，如果除了待排序对象自身所占的空间之外，排序算法只利用常量内存空间进行排序，则称这个排序算法是原位（in-place）排序。归并排序算法（正如上述描述的那样），不是原位排序算法，要使它成为原位排序算法，需要比4.1.1节讨论的更复杂的归并方法。不过，原位排序本质上并不困难，因为堆排序、快速排序都适合于原位排序。

但是，把快速排序算法变成原位排序算法需要一点智慧。必须利用输入序列自身存储所有递归调用中的子序列。算法inPlaceQuickSort进行原位快速排序，如算法4-4所示。算法inPlaceQuickSort假设输入序列 S 中的元素互不相同。实施这种限制的原因在习题R-4.12中进行了探讨。习题C-4.18中对扩展到一般情况进行了讨论。算法基于位序方法访问输入序列 S 中的元素。因此，假如用数组实现 S ，则算法将有效运行。

算法4-4 原位快速排序对用数组实现的序列进行排序

```

算法 inPlaceQuickSort( $S, a, b$ ):
    输入: 不同元素组成的序列 $S$ ，及整数 $a$ 和 $b$ 
    输出: 按位序从 $a$ 到 $b$ （含 $a$ 和 $b$ ）非降序输出有序序列 $S$ ， $S$ 中的元素最初是按位序从 $a$ 到 $b$ 存放的。
    if  $a \geq b$  then return {子区间为空}
     $p \leftarrow S.\text{elemAtRank}(b)$  {枢轴元素}
     $l \leftarrow a$  {将向右扫描}
     $r \leftarrow b - 1$  {将向左扫描}
    while  $l \leq r$  do
        {找出大于枢轴元素的元素}
        while  $l \leq r$  and  $S.\text{elemAtRank}(l) \leq p$  do
             $l \leftarrow l + 1$ 
        {找出小于枢轴元素的元素}
        while  $r \geq l$  and  $S.\text{elemAtRank}(r) \geq p$  do
             $r \leftarrow r - 1$ 
        if  $l < r$  then
             $S.\text{swapElements}(S.\text{atRank}(l), S.\text{atRank}(r))$ 
        {把枢轴元素放入其最终位置}
         $S.\text{swapElements}(S.\text{atRank}(l), S.\text{atRank}(b))$ 
        {递归调用}
        inPlaceQuickSort( $S, a, l - 1$ )
        inPlaceQuickSort( $S, l + 1, b$ )
  
```

248

原位快速排序利用swapElements操作修改输入序列，它并没有显式地创建子序列。输入序列的子序列用一个位置区间隐含表示，该区间的最左边位序是 l ，最右边位序是 r 。通过从 l 向前以及从 r 向后同时扫描，执行划分步骤。并交换那些处于逆序的元素对。如图4-13所示。当这两个下标“相遇”时，子序列 L 和 G 位于相遇点相反的两边。通过对两个子序列进行递归，算法完成执行过程。

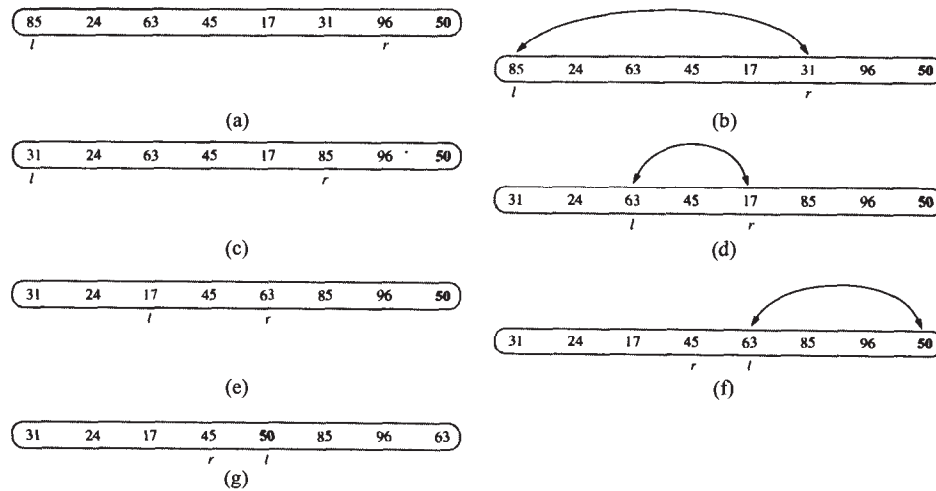


图4-13 原位快速排序的划分步骤。下标 l 从左向右扫描序列，下标 r 从右向左扫描序列。当 l 所在元素大于枢轴元素并且 r 所在元素小于枢轴元素时，则进行一次交换。最终与枢轴元素的交换完成划分步骤

原位快速排序算法通过建立新序列，并在序列之间移动元素，使算法运行时间减少了一个常数因子。代码段4-1显示了原位快速排序的Java版本。

代码段4-1 原位快速排序的Java实现。它假设用数组实现输入序列，并且输入序列中的元素互不相同

```
/**
 * Sort the elements of sequence S in nondecreasing order according
 * to comparator c, using the quick-sort algorithm. Most of the work
 * is done by the auxiliary recursive method quickSortStep.
 */
public static void quickSort (Sequence S, Comparator c) {
    if (S.size() < 2)
        return; // a sequence with 0 or 1 element is already sorted
    quickSortStep(S, c, 0, S.size()-1); // recursive sort method
}

/**
 * Sort in nondecreasing order the elements of sequence S between
 * ranks leftBound and rightBound, using a recursive, in-place,
 * implementation of the quick-sort algorithm.
 */
private static void quickSortStep (Sequence S, Comparator c,
                                   int leftBound, int rightBound) {
    if (leftBound >= rightBound)
        return;
    Object pivot = S.atRank(rightBound).element();
    int leftIndex = leftBound; // will scan rightward
    int rightIndex = rightBound-1; // will scan leftward
    while (leftIndex <= rightIndex) {
        // scan rightward to find an element larger than the pivot
        while ( (leftIndex <= rightIndex) &&
```

```

        c.isLessThanOrEqualTo(S.atRank(leftIndex).element(), pivot) )
    leftIndex++;
    // scan leftward to find an element smaller than the pivot
    while ( (rightIndex >= leftIndex) &&
        c.isGreaterThanOrEqualTo(S.atRank(rightIndex).element(), pivot) )
        rightIndex--;
    if (leftIndex < rightIndex) // both elements were found
        S.swapElements(S.atRank(leftIndex), S.atRank(rightIndex));
} // the loop continues until the indices cross
// place the pivot by swapping it with the element at leftIndex
S.swapElements(S.atRank(leftIndex), S.atRank(rightBound));
// the pivot is now at leftIndex, so recur on both sides of it
quickSortStep(S, c, leftBound, leftIndex-1);
quickSortStep(S, c, leftIndex+1, rightBound);
}

```

249
?
250

不幸的是，从技术上说，我们实现的快速排序不完全是原位排序，因为它需要的空间比常量空间要多。子序列并没有使用额外空间，只有局部变量（如 l 和 r ）使用了常量空间。那么那些额外的空间来自何处？它来自递归过程，回忆2.1.1节，栈所需的空间与快速排序递归树的深度成正比。这个深度至少为 $\log n$ ，至多为 $n-1$ 。为了使快速排序真正成为原位排序，必须用非递归方式实现它（且不用栈）。这样实现的关键细节是需要一种原位方式，确定“当前”子序列左、右边界的界限。然而，这样的模式并不太难，我们把实现的细节留作习题（C-4.17）。

4.9 习题

基础题

- R-4.1 给出定理4.1的完整证明。
- R-4.2 给出归并排序算法的伪代码描述。可以调用merge算法作为子例程。
- R-4.3 给出归并排序算法一个变体的伪代码描述，它在数组上而不是在一般序列上执行操作。
- R-4.4 证明归并排序算法对 n 个元素的序列进行排序的运行时间为 $O(n \log n)$ ，即使 n 不是2的幂，结论仍然成立。
- R-4.5 给定两个 n 个元素的有序序列 A 和 B ，不能把它们看作集合（即 A 和 B 可能包含相同的元素）。描述一个 $O(n)$ 时间的方法，计算表示集合 $A \cup B$ 的序列（没有相同的元素）。
- R-4.6 对于任意三个集合 X 、 A 和 B ，证明 $(X - A) \cup (X - B) = X - (A \cap B)$ 。
- R-4.7 假定只利用按大小合并的启发式搜索，实现基于树的划分（合并寻找（union-find））数据结构。在这种情况下， n 个union和find操作的序列的平摊运行时间是多少？
- R-4.8 编写伪代码，描述在一个有序序列实现的集合上执行insert和remove方法。
- R-4.9 假定修改确定性快速排序算法，使它不是选择 n 个元素中的最后一个元素作为枢轴元素，而是选择位序（索引）为 $\lfloor n/2 \rfloor$ 的元素作为枢轴元素，即序列中间的元素。对于一个已经有序的序列，这个版本的快速排序的运行时间是多少？
- R-4.10 再次考虑确定性快速排序算法的修改版本。它不是选择 n 个元素中的最后一个元素作为枢轴元素，而是选择位序为 $\lfloor n/2 \rfloor$ 的元素作为枢轴元素。描述一种序列，使得该版本的快速排序算法的运行时间为 $\Theta(n^2)$ 。
- R-4.11 对于 n 个不同元素组成的序列，证明快速排序最好情况下的运行时间为 $O(n \log n)$ 。
- R-4.12 假定在含有相同元素的序列上执行算法inPlaceQuickSort（算法4-4）。证明在这种情况下，算法对输入序列排序正确。但是划分步骤的结果可能不同于4.3节给出的高级描述，导致效率不高。特别是当存在元素与枢轴元素相等时，划分步骤中会发生什么情况？实际上计算序列 E （存储等于枢轴元素的元素）？算法会对子序列 L 和 R 执行递归吗？或者对其他子序列执行递归吗？如果输入序列中的所有元素都相同，算法的运行时间是多少？

251

- R-4.13 给出原位快速排序的伪代码描述，它以数组作为输入，而不是以序列作为输入，并返回与输出相同的数组。
- R-4.14 对于冒泡排序、堆排序、归并排序和快速排序，其中哪些（如果有的话）是稳定的排序算法？
- R-4.15 描述一个基数排序方法，对三元组 (k, l, m) 组成的序列 S 进行字典排序，其中 k, l, m 是 $[0, N-1]$ 区间内的整数， $N \geq 2$ 。如何扩展这个模式，使其适合于 d 元组 (k_1, k_2, \dots, k_d) ，其中每个 k_i 是 $[0, N-1]$ 区间内的整数。
- R-4.16 桶排序算法是原位排序算法吗？为什么？
- R-4.17 给出原位快速选择算法的伪代码描述。
- R-4.18 证明作用在 n 个元素的序列上的快速选择算法最坏情况下的运行时间为 $\Omega(n^2)$ 。

创新题

- C-4.1 假设集合 A 和 B 是用有序序列实现的，借助泛型归并算法的一个变体，证明如何实现集合 A 上的方法 $\text{equals}(B)$ ，使其测试 $A=B$ 是否相等的时间为 $O(|A| + |B|)$ 。
- C-4.2 给出泛型归并算法的一个变体，用于计算 $A \oplus B$ ，使其包含 A 中的元素或者 B 中的元素，但不包含这两个集合中共有的元素。
- C-4.3 假定用平衡查找树表示每个集合，实现集合ADT。描述并分析用于集合ADT中的每个方法的算法。
- C-4.4 设 A 是对象集合。描述一个有效的方法，用于将 A 转换成一个集合，即从 A 中删除所有重复的元素，这个方法的运行时间是多少？
- C-4.5 考虑其元素是区间 $[0, N-1]$ 内的整数的集合。表示这类集合 A 的流行模式是利用一个布尔向量 B ，当且仅当 $B[x] = \text{true}$ 时，称 x 在 A 中。因为 B 的每一个单元可用一位表示，所以有时称 B 为位向量（bit vector）。假定采用所述的集合表示方法，描述执行集合中的union、intersection和subtraction方法的有效算法，并分析这些方法的运行时间。
- C-4.6 假定利用按大小合并的启发式方法以及局部（partial）路径压缩启发式方法，实现基于树的划分（合并寻找）数据结构。这种情况的局部路径压缩的含义是，在进行find操作的一系列指针跳跃之后，沿着指向其祖父结点的这条路径，更新每个结点 u 的parent指针。证明执行 n 个union和find操作的总运行时间仍然为 $O(n \log^* n)$ 。
- C-4.7 假定利用按大小合并的启发式方法和路径压缩启发式方法，实现基于树的划分（合并寻找）数据结构。如果所有union出现在find之前，证明执行 n 个union和find操作的总运行时间为 $O(n)$ 。
- C-4.8 证明随机化快速排序的运行时间为 $O(n \log n)$ 的概率是 $1 - 1/n^2$ 。
提示：利用Chernoff界限，它指出如果投掷一枚硬币 k 次，那么出现正面次数少于 $k/16$ 的概率小于 $2^{-k/8}$ 。
- C-4.9 给定 n 个元素的序列 S ，每个元素着以红色或蓝色。假设 S 表示为数组，给出一种对 S 的原位排序方法，满足所有蓝色元素都列在所有红色元素之前。可将你的方法扩展到三种颜色吗？
- C-4.10 给定 n 个元素的序列 S ，满足 S 中的每个元素表示一次选举中的不同选票，每张选票被给出为一个整数，表示所选候选人的ID。不做谁在竞选或者有多少名候选人的假设，设计一个 $O(n \log n)$ 时间的算法，查明谁将赢得 S 表示的选举。假设拥有最多选票的候选人获胜。
- C-4.11 考虑上一个问题中的选举问题，但现在假设有 $k < n$ 名候选人参选。描述一个 $O(n \log k)$ 时间的算法，确定谁将赢得选举。
- C-4.12 证明在不影响算法渐近运行时间的前提下，任何基于比较的排序算法都可变成稳定的排序算法。
提示：改变元素相互比较的方式。
- C-4.13 给定两个具有 n 个元素的序列 A 和 B ，它们可能包含相同的元素，在这些元素上定义了全序关系。描述一个确定 A 和 B 是否包含相同元素集合（次序可能不同）的有效算法，这一方法的运行时间是多少？
- C-4.14 给定 n 个元素的序列 S ，其中每个元素是 $[0, n^2-1]$ 区间内的整数。描述一个简单的方法，用于在 $O(n)$ 时间内对 S 排序。
- C-4.15 设 S_1, S_2, \dots, S_k 是 k 个不同的序列，它们的每个元素具有 $[0, M]$ 区间内的关键字，其中参数 $N \geq 2$ 。描述一个对所有序列（不是作为集合的并）进行排序且运行时间为 $O(n + M)$ 的算法，其中 n 表示所有序列的总大小。

C-4.16 给定 n 个元素的序列 S ，在这些元素上定义了全序关系。描述一个有效的方法，确定 S 中是否存在两个相同的元素。你的方法的运行时间是多少？

C-4.17 描述一个非递归、原位快速排序算法。该算法仍然应该基于相同的分治法。

提示：在对一个子序列进行递归调用之前，考虑如何“标记”当前子序列的左、右边界。

253 C-4.18 修改算法inPlaceQuickSort（算法4-4），使其能够有效处理输入序列 S 中含有相同元素的一般情形。

C-4.19 设 S 是 n 个元素的序列，并定义了其上的全序关系。 S 中的逆（inversion）是 x 和 y 的元素对，满足在 S 中， x 出现在 y 之前，但 $x > y$ 。描述一个运行时间为 $O(n \log n)$ 的算法，确定 S 中的逆序个数。

提示：考虑修改归并排序算法，求解此问题。

C-4.20 设 S 是 n 个元素的序列，并定义了其上的全序关系。描述一个基于比较的方法，它对 S 进行排序的时间为 $O(n + k)$ ，其中 k 是 S 中的逆序个数（回忆上一个问题中关于逆序的定义）。

提示：考虑原位插入排序算法，在经过线性时间的预处理步骤后，只交换那些呈逆序的元素。

C-4.21 给出一个 n 个整数的序列，其中含有 $\Omega(n^2)$ 个逆序（由习题C-4.19可知逆序的定义）。

C-4.22 设 A 和 B 都是具有 n 个整数的两个序列。给出一个整数 x ，描述一个 $O(n \log n)$ 时间的算法，确定 A 中是否存在整数 a ， B 中是否存在整数 b ，满足 $x = a + b$ 。

C-4.23 给定 n 个可比较元素的无序序列 S ，描述一个有效算法，找出 $\lceil \sqrt{n} \rceil$ 个数据项，满足其在有序序列 S 中的位序与中值的位序最接近。你的方法的运行时间是多少？

C-4.24 这个问题是修改快速选择算法，使之成为确定性算法，同时对于 n 个元素的序列，运行时间仍为 $O(n)$ 。其思想是修改选择枢轴元素的方式，使它能够进行确定性选择，而不是随机性选择。如下：

把集合 S 划分成每个大小均为5个元素的 $\lceil n/5 \rceil$ 个组（可能有一组不满5个元素）。对每个小集合进行排序，并找出这个集合中的中值元素。从 $\lceil n/5 \rceil$ 个“子”中值这个集合中，递归利用选择算法找出这些子中值的中值。利用这个中值元素作为枢轴元素，并像执行快速选择算法一样。

通过回答下列问题证明这个确定性算法的运行时间为 $O(n)$ （在简化数学表达式时，请忽略向下取整函数及向上取整函数，因为对于渐近表示，不论哪种情况结果都一样）。

- 有多少个子中值小于或等于所选的枢轴元素？有多少个子中值大于或等于所选的枢轴元素？
- 对于每个小于或等于枢轴元素的子中值，有多少个其他元素小于或等于这个枢轴元素？对于那些大于或等于这个枢轴元素的元素同样如此吗？
- 证明为什么找出确定性枢轴元素并用它对 S 进行划分的方法的运行时间为 $O(n)$ 。
- 基于这些估计，编写一个递归方程，界定这个选择算法最坏情况下的运行时间 $t(n)$ （注意：在最坏情况下，有两个递归调用——一个是找出子中值的中值，另一个是对 L 和 G 中的较大者进行递归）。

254 e. 利用这个递归方程，用归纳法证明 $t(n)$ 是 $O(n)$ 。

C-4.25 Bob有一个具有 n 个螺帽的集合 A 和一个具有 n 个螺母的集合 B ，满足 A 中的每个螺帽唯一匹配 B 中的一个螺母。不幸的是， A 中的所有螺帽看起来是一样的， B 中的所有螺母也看起来是一样的。Bob所能做的唯一一类比较是取螺帽-螺母 (a, b) 对，满足 $a \in A$ 且 $b \in B$ ，测试 a 的螺纹数是否多于、少于或完全匹配 b 的螺纹数。为Bob描述一个有效算法，使他的所有螺帽和螺母相匹配。根据Bob执行的螺帽-螺母测试次数，给出该算法的运行时间。

C-4.26 证明如何把一个 $O(n)$ 时间的确定性选择算法用于设计一个像快速排序那样的排序算法，对于 n 个元素的序列，最坏情况下的运行时间为 $O(n \log n)$ 。

C-4.27 给定 n 个可比较元素的无序序列 S 和一个整数 k ，给出一个 $O(n \log k)$ 的期望时间的算法，用于找出位序为 $\lceil n/k \rceil$ 、 $2\lceil n/k \rceil$ 、 $3\lceil n/k \rceil$ 等的 $O(k)$ 个元素。

程序设计

P-4.1 设计和实现一个 $O(n + N)$ 运行时间的稳定桶排序算法，对 n 个元素的序列进行排序，序列中每个元素具有的整型关键字取自区间 $[0, N-1]$ ，其中 $N \geq 2$ 。执行一系列基准时间试验，测试对于各种不同的 n 和 N 值，这个方法的运行时间是否确实为这个时间，编写一份简短的报告，描述代码及这些试验的结果。

- P-4.2 实现归并排序和确定性快速排序算法，执行一系列基准测试，表明哪个算法更快。你的测试中应该包括那些看起来是“随机”的序列，以及那些“几乎有序”或“几乎逆序”的序列。编写一份描述代码及这些试验的结果的简短报告。
- P-4.3 实现确定性快速排序算法和随机化快速排序算法，执行一系列基准测试，表明哪个算法更快。你的测试中应该包括那些看起来是“随机”的序列，以及那些“几乎”有序的序列。编写一份描述代码及这些试验的结果的简短报告。
- P-4.4 实现原位插入排序算法和原位快速排序算法，执行一系列基准测试，确定 n 值的范围，使得平均而言快速排序快于插入排序。
- P-4.5 设计和实现本章描述的排序算法之一的动画。你的动画应该直观地说明该算法中关键字的性质，应该加注文本和/或声音，用以向某些不熟悉此算法的人做出解释。编写一份描述该动画的简短报告。
- P-4.6 利用基于树的方法，结合按大小合并及路径压缩的启发式方法，实现划分（合并寻找）ADT。

255

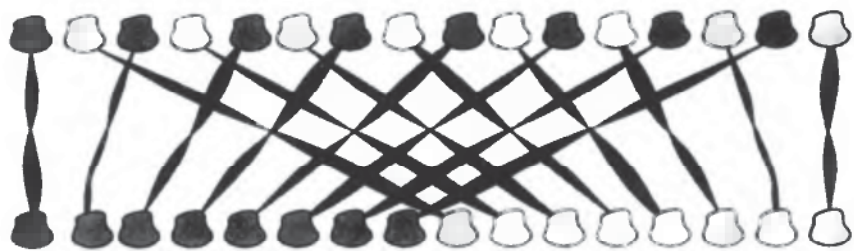
4.10 本章注记

Knuth的经典著作Sorting and Searching[119]包含了排序问题及其解法的广泛历史，从19世纪后期的人口普查卡排序机开始。Huang和Langston[103]描述了如何在线性时间内原位归并两个有序表。本章中的集合ADT源于Aho、Hopcroft和Ullman[8]的集合ADT。标准快速排序算法起源于Hoare[96]。随机化快速排序的更紧致分析可在Motwani和Raghavan的著作[157]中找到。Gonnet和Baeza-Yates[81]提供了对不同排序算法的实验及理论上的比较。术语“剪枝-查找”源于计算几何中的文献（如Clarkson[47]和Megiddo[145、146]）。术语“减小-求解”来自于Levitin[132]。

对划分数据结构的分析来自于Hopcroft和Ullman[99]（也可见[7]）。Tarjan[199]证明了对于 n 个union和find操作的序列，本章描述的实现执行时间为 $O(n\alpha(n))$ ，其中 $\alpha(n)$ 是慢速增长的Ackermann逆函数。最坏情况下，这个界限是紧致的（也见[200]）。但是，Gabow和Tarjan[73]证明，在某些情况下，其运行时间可达 $O(n)$ 。

256

第5章



基本技术

有一个流行的电视网播出了两个不同的木工制作节目。在一个节目中，主人利用特制强力工具制作家具；在另一个节目中，主持人利用通用手工工具制作家具。第一个节目中使用的特制工具很适合专门的用途，但这些工具不通用。但是，第二个节目中的工具是基本工具，因为它们能够有效地用来解决一大类广泛不同的任务。

这两个电视节目提供了数据结构和算法设计的一个有趣的隐喻。有些算法工具相当专业，只能用于某些特定问题，不具通用性。然而，还有一些基本的（fundamental）算法工具，可用于广泛不同的数据结构和算法设计问题中。学习利用这些基本技术是一门艺术，本章专门介绍有效利用这些技术的知识。

本章中介绍的基本技术包括贪心法、分治法和动态规划。这些技术具有通用性，本章以及本书其他一些章中给出了这些技术应用的例子。

贪心算法用于加权图的算法中，将在第7章中讨论。它还用于9.3节介绍的数据压缩问题中。顾名思义，这项技术的主要思想是执行一系列的贪心选择，以便构造给定问题的一个最优解。在这一章里，给出了贪心算法的一般结构，并表明如何将这项技术用于背包问题和调度问题。

分治法用于第4章中讨论的归并排序和快速排序算法中。这项技术的一般思想是将给定问题分解成许多类似的子问题，并递归求解每个子问题，直到这些子问题小得足以直接用蛮力（brute force）方法求解，递归调用返回之后，将所有子问题的解归并为原问题的解。在这一章里，表明如何设计和分析一般的分治算法，并将这项技术用于大整数相乘和大矩阵相乘的问题中。同时给出了大量求解分治递归方程的技术，这些技术包括一般主定理，它可用于各种方程中。

动态规划技术初看起来可能有点令人迷惑，但它是一项相当强人的技术。其主要思想是利用一组较小的整数下标表征子问题，来解决给定的问题。表征的目标是允许通过组合更小子问题（可能有重叠）的解来定义较大子问题的最优解。如果能够构造这样一个表征，我们就能建立由较小子问题构造更大子问题解法的相当直观的算法，这是利用动态规划技术中最困难的一步。这项技术是第6章中Floyd-Warshall传递闭包算法的基础。在这一章里，我们描述了动态规划的一般框架，给出了几个应用的例子，包括0-1背包问题。

258

5.1 贪心法

本章考虑的第一种算法设计技术是贪心法（greedy method）。根据一般贪心选择的性质，表

征这种贪心算法的设计模式，并给出了它的两个应用。

贪心算法应用于优化问题，即问题涉及通过一组配置（configuration）来找出定义在这些配置上的目标函数（objective function）的最小值或最大值。贪心算法的一般表述可能不太简单。为了求解给定的优化问题，进行一系列选择。这个序列开始于某些易于理解的起始配置，然后从当前可能的配置中，反复做出看起来是最好的决策。

这个贪心算法并不会总是导致问题的最优解。但有几个问题采用此方法确实会得到最优解。称这样的问题具有贪心选择（greedy-choice）性质。这个性质说明，从一个良好定义的配置开始，通过一系列局部最优选择（即基于当时可用的可能配置做出最佳选择），可以得到全局最优配置。

5.1.1 背包问题

考虑背包（fractional knapsack）问题，给定 n 个物品的集合 S ，满足每个物品 i 都有一个正效用 b_i 及一个正权值 w_i ，希望找出具有最大效用的子集，同时它不超过一个给定的权值 W 。如果将每个物品限制为要么完全接受，要么拒绝，则得到0-1背包问题（5.3.3节给出了这一问题的动态规划解）。但是，现在允许取某些元素的任意一部分。背包问题的动机是，我们准备一次旅行，只有一个背包，所能携带物品的总权值至多为 W 。此外，可将物品分成任意部分。也就是说，可取每个物品 i 的数量 x_i ，满足

$$\text{对于每个 } i \in S, 0 \leq x_i \leq w_i, \sum_{i \in S} x_i \leq W$$

物品的总效用由目标函数

$$\sum_{i \in S} b_i(x_i / w_i)$$

确定。

考虑某个学生准备进行户外运动的事件，学生必须用食物填充背包。每种候选食物可容易地分解成若干部分，如汽水、土豆片、爆米花和比萨饼。

这是贪心法成功应用的一个例子。因为利用算法5-1所示的贪心算法可以求解背包问题。

259

算法5-1 背包问题的贪心算法

算法 FractionalKnapsack(S, W):

输入: 物品集合 S ，满足对于每个物品 $i \in S$ ，都有一个正效用 b_i 及一个正权值 w_i ；正最大总权值 W

输出: 每个物品 $i \in S$ 的数量 x_i ，使总效用最大，同时不超过最大总权值 W

for 每个物品 $i \in S$ **do**

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$ {物品 i 的值下标}

$w \leftarrow 0$ {总权值}

while $w < W$ **do**

 从 S 中删除一个具有最大值下标的物品 i {贪心选择}

$a \leftarrow \min\{w_i, W-w\}$ {超过 $W-w$ 将会引起权值溢出}

$x_i \leftarrow a$

$w \leftarrow w + a$

FractionalKnapsack算法可在 $O(n \log n)$ 时间内实现，其中 n 是 S 中的物品数量。确切地讲，可以利用基于堆的优先队列（2.4.3节）存储 S 中的物品，其中每个物品的关键字就是它的值下标。利用这个数据结构，每次贪心选择（用于删除具有最大值下标的物品）所需时间为 $O(\log n)$ 。

为了表明背包问题满足贪心选择的性质, 假定存在两个物品 i 和 j , 满足

$$x_i < w_i, x_j > 0 \text{ 且 } v_i > v_j$$

令

$$y = \min\{w_i - x_i, x_j\}$$

可以取物品 i 的等量部分代替物品 j 的数量 y , 因此提高了总效用, 而不会改变总权值。于是, 可以通过贪心选择具有最大值下标的物品, 正确地计算物品的最优数量。导致如下定理。

定理5.1 给定 n 个物品的集合 S , 满足每个物品 i 都有一个效用 b_i 及一个权值 w_i , 可以用 $O(n \log n)$ 时间构造 S 的一个最大效用子集, 允许取一部分物品, 且不超过总权值 W 。

该定理表明可以有效地求解背包问题。然而, 所有-或-没有或者“0-1”背包问题不满足这个贪心选择的性质, 求解这个问题要困难得多, 将在5.3.3节和13.3.4节探讨。

260

5.1.2 任务调度

考虑另一个优化问题。给定 n 个任务的集合 T , 满足每个任务 i 有一个开始时间 (start time) s_i 和一个完成时间 (finish time) f_i (其中 $s_i < f_i$)。每项任务必须在一台机器上执行, 每台机器一次只能执行一项任务。如果两项任务 i 和 j 满足 $f_i \leq s_j$ 或 $f_j \leq s_i$, 则称这两项任务是不冲突的 (nonconflicting)。仅当两项任务不冲突时, 才可在同一台机器上调度执行它们。

这里考虑的任务调度 (task scheduling) 问题是, 以不冲突方式用尽可能少的机器调度 T 中的所有任务。可以换一种想法, 将任务看作会议, 必须用尽可能少的会议室调度所有会议。如图5-1所示。

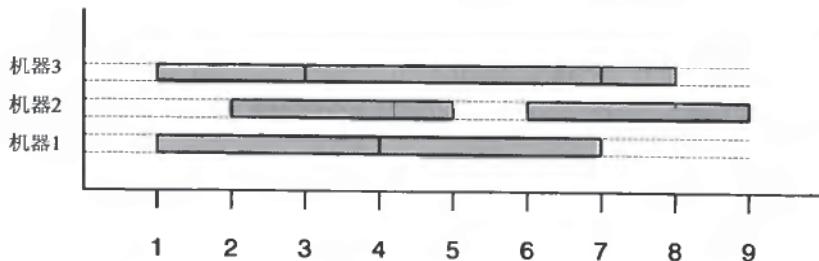


图5-1 任务调度问题求解的说明, 各项任务开始时间和完成时间对的集合为 $\{(1, 3), (1, 4), (2, 5), (3, 7), (4, 7), (6, 9), (7, 8)\}$

在算法5-2中, 描述了用于此问题的一个简单的贪心算法。

算法5-2 用于任务调度问题的贪心算法

算法 TaskSchedule(T):

输入: 任务集合 T , 满足每个任务 i 有一个开始时间 s_i 和一个完成时间 f_i

输出: 利用最少数量的机器, 输出 T 中不冲突的任务调度

$m \leftarrow 0$ {最优机器数}

while $T \neq \emptyset$ **do**

从 T 中删除具有最小开始时间 s_i 的任务 i

if 存在机器 j , 其上执行的任务与任务 i 不冲突 **then**

在机器 j 上调度任务 i

else

$$m \leftarrow m+1 \quad \{\text{添加一台新机器}\}$$

在机器 m 上调度任务 i

261

贪心任务调度算法的正确性

在TaskSchedule算法中,开始时没有机器,并按照开始时间的次序,以贪心方式考虑任务。对于每项任务 i ,如果存在可以处理任务 i 的机器,则在那台机器上调度 i 。否则,分配一台新机器,并在其上调度任务 i 。重复这个贪心选择的过程,直到考虑了 T 中所有的任务。

定理5.2确定了上述TaskSchedule算法将正确工作的事实。

定理5.2 给定 n 个任务的集合,每项任务都被指定了一个开始时间和一个完成时间。算法TaskSchedule产生利用最少数量的机器调度这些任务所需的时间为 $O(n \log n)$ 。

证明 利用简单的反证法,可以证明上述简单的贪心算法TaskSchedule,用最少数量的机器找出一种最优调度。■

假定算法没有找出最优调度,即假定算法利用 k 台机器,找出一个非冲突的调度,但还存在另一个利用 $k-1$ 台机器的非冲突调度。设 k 是算法分配的最后一台机器, i 是在 k 上调度的第一项任务。由算法的结构可知,当调度任务 i 时,在 $1 \sim (k-1)$ 的每台机器中包含的任务与任务 i 冲突。由于这些任务与 i 冲突,又因为我们按照开始时间考虑任务,所以当前与任务 i 冲突的那些任务的开始时间必定小于或等于 i 的开始时间 s_i ,且它们的完成时间在 s_i 之后。换句话说,这些任务不仅与任务 i 冲突,它们还相互冲突。但是,这意味着在集合 T 中,有 k 项任务相互冲突,这蕴涵着只利用 $k-1$ 台机器不可能调度 T 中的所有任务。于是, k 是调度 T 中所有任务所需的最少机器数。

证明如何用 $O(n \log n)$ 时间实现算法TaskSchedule的工作留作习题(R-5.2)。

考虑本书中贪心法的几个其他的应用,包括串压缩中的两个问题(9.3节),其中贪心法引出的构造称为Huffman编码;还包括图算法(7.3节),贪心法用于求解最短路径问题和最小生成树问题。

讨论的下一技术是分治法,这是一个利用递归设计有效算法的一般方法。

262

5.2 分治法

分治技术涉及求解特殊的计算问题,它通过把问题划分成一个或多个更小的子问题,递归地求解每个子问题,然后通过“归并”或“结合”子问题的解,产生原问题的解。

可用原问题规模 n 作为参数对分治法建模,设 $S(n)$ 表示这个问题。通过求解 k 个子问题 $S(n_1), S(n_2), \dots, S(n_k)$ 的集合,并归并这些子问题的解得到原问题的解,其中 $n_i < n, i = 1, 2, \dots, k$ 。

例如,在经典的归并排序算法中(4.1节), $S(n)$ 表示对 n 个数的序列排序的问题。归并排序将 $S(n)$ 分成两个子问题 $S(\lfloor n/2 \rfloor)$ 和 $S(\lceil n/2 \rceil)$,递归求解这两个子问题,然后把得到的有序序列归并为一个有序序列,产生 $S(n)$ 的一个解。归并步骤所需时间为 $O(n)$,归并排序算法的总运行时间为 $O(n \log n)$ 。

就像归并排序算法一样,用一般分治技术建立的算法具有快速运行时间。

5.2.1 分治递归方程

我们利用递归方程(recurrence equation)(1.1.4节)分析分治算法的运行时间,即设 $T(n)$ 表示输入大小为 n 的算法的运行时间,利用一个方程表征 $T(n)$,该方程将 $T(n)$ 与针对小于 n 的问题规

模的函数 T 的值关联起来, 对于归并排序算法, 递归方程如下:

$$T(n) = \begin{cases} b & \text{如果 } n < 2 \\ 2T(n/2) + bn & \text{如果 } n \geq 2 \end{cases}$$

其中常数 $b > 0$, 假设 n 为2的幂。事实上, 在这一节里, 为了简化, 都假设 n 为某个合适的幂, 由此, 可以避免利用向下取整函数和向上取整函数。即使放宽这个假设, 所给出的每个渐近声明仍然为真。但这个事实形式化的论证过程涉及又长又枯燥的证明。如上所观察的, 可以证明这种情况下 $T(n)$ 为 $O(n \log n)$ 。但是, 一般而言, 可能得到的递归方程要比这个方程更具挑战性。因此, 它可用于开发求解各类递归方程的一些常规方式, 这些递归方程通常出现在分治算法的分析中。

[263]

1. 迭代代换法

求解分治递归方程的一种方法是迭代代换法 (iterative substitution method)。口头上常称之为“plug-and-chug”法。在应用这种方法时, 假设问题规模 n 相当大, 并在方程右边出现 T 的每个地方代入递归的一般形式。例如, 在归并排序的递归方程中进行这样的代入, 可得方程

$$\begin{aligned} T(n) &= 2(2T(n/2^2) + b(n/2)) + bn \\ &= 2^2T(n/2^2) + 2bn \end{aligned}$$

再次插入 T 的一般形式, 得到方程

$$\begin{aligned} T(n) &= 2^2(2T(n/2^3) + b(n/2^2)) + 2bn \\ &= 2^3T(n/2^3) + 3bn \end{aligned}$$

在应用迭代代换法时, 希望在某个点上看到模式能够被转换成一般的封闭形式的方程 (T 只出现在方程的左边)。对于归并排序递归方程的情况, 一般形式是

$$T(n) = 2^i T(n/2^i) + ibn$$

注意此方程一般形式的递推基础为 $T(n) = b$ 。当 $n = 2^i$ 即 $i = \log n$ 时, 蕴涵着

$$T(n) = bn + bn \log n$$

换句话说, $T(n)$ 是 $O(n \log n)$ 。在迭代代换技术的一般应用中, 希望能够确定 $T(n)$ 的一般模式, 同时还能查明何时 $T(n)$ 的一般形式递推到基础形式。

从数学的观点看, 在利用迭代代换技术的过程中, 在某一点上涉及一点逻辑“跳跃”。在试图通过代入序列表征一般形式时会出现这种跳跃。对于归并排序递归方程的情形, 这种跳跃是十分合理的。但是, 在其他一些时间, 方程的一般形式会是什么样子的并不明显。在这些情况下, 跳跃可能更危险。为了使所执行的跳跃完全安全, 必须充分证明方程的一般形式, 可能利用归纳法。结合这样的证明技术, 迭代代换方法是完全正确的, 并且它是表征递归方程的常用方式。顺便说一句, 用于描述迭代代换法的口语“plug-and-chug”源于: “plugging”涉及方程 $T(n)$ 的递归部分, “chugging”涉及相当数量的代数计算, 以便把方程变成能够推理一般模式的形式。

[264]

2. 递归树

另一种表征递归方程的方法是使用递归树 (recursion tree) 方法。如同迭代代换法, 这种技术使用反复代入求解递归方程。但它不同于迭代代换法, 这在于它不是一种代数方法, 而是一种可视化方法。在利用递归树方法时, 画一棵树 R , 其中每个结点表示递归方程的一个不同的代入。因此, R 中的每个结点都有一个与之关联的函数 $T(n)$ 的参数 n 的值。此外, R 中的每个结点 v 都关联一个开销 (overhead), 定义为结点 v 的递归方程中非递归部分的值。对于分治递归方程, 开销对应于归并来自于 v 的子结点的子问题的解所需的运行时间。对关联 R 中所有结点的开销求和, 得到递归方程的解。普遍的做法是, 首先对 R 中每一层的值求和, 然后再对 R 中所有层的部分和求和。

示例5.1 考虑如下递归方程:

$$T(n) = \begin{cases} b & \text{如果 } n < 3 \\ 3T(n/3) + bn & \text{如果 } n \geq 3 \end{cases}$$

这是我们得到的递归方程,例如,通过修改归并排序算法,将一个无序序列划分成三个大小相等的序列,对每个序列递归进行排序,然后对三个有序序列进行三路归并,产生原序列的一个有序版本。在这个方程对应的递归树 R 中,每个内部结点 v 有三个子结点,并且具有一个大小和一个与之关联的开销,它对应于归并 v 的子结点所产生的子问题的解所需的时间。图5-2说明了树 R 。注意每层结点的开销之和为 bn 。因此,观察可见, R 的深度为 $\log_3 n$,因而 $T(n)$ 是 $O(n \log n)$ 。

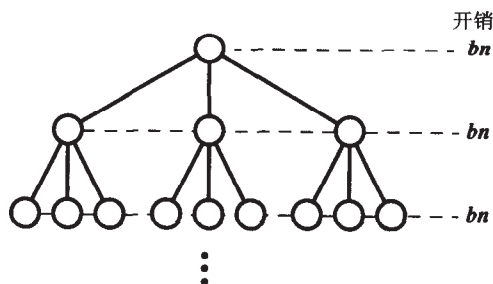


图5-2 示例5.1中的递归树 R , 其中显示了每一层累积的开销

265

3. 猜测-测试法

另一种求解递归方程的方法是猜测-测试(guess-and-test)技术。这项技术首先对递归方程可能具有的封闭形式做出有根据的猜测,然后利用归纳法证明这个猜测。例如,可以利用猜测-测试法作为一种“二分搜索”法,找出给定递归方程的一个良好上界。如果对当前猜测的证明失败,那么可能需要利用一个增长更快的函数,如果当前猜测证明“太容易”,那么可能需要一个增长更慢的函数。然而,利用这项技术时要非常小心,在采取的每个数学步骤中,都尝试证明某个假设关于当前“猜测”成立。我们探索猜测-测试法在以下例子中应用。

示例5.2 考虑如下递归方程(假设基本情况为 $T(n) = b$, 其中 $n < 2$):

$$T(n) = 2T(n/2) + bn \log n$$

这个方程看起来与归并排序例程的递归方程非常类似。首先可能会做出以下猜测:

$$\text{首次猜测: } T(n) \leq cn \log n$$

其中常数 $c > 0$ 。可以选择足够大的 c ,使猜测对于基本情况成立,因此,只考虑 $n \geq 2$ 的情况。如果首先所做的猜测是,归纳假设对于输入大小小于 n 成立,那么可得

$$\begin{aligned} T(n) &= 2T(n/2) + bn \log n \\ &\leq 2(c(n/2) \log(n/2)) + bn \log n \\ &= cn(\log n - \log 2) + bn \log n \\ &= cn \log n - cn + bn \log n \end{aligned}$$

但是对于 $n \geq 2$,不存在任何方法使得上式最后一行小于或等于 $cn \log n$ 。因此,第一个猜测是不充分的。再试

$$\text{更好的猜测: } T(n) \leq cn \log^2 n$$

其中常数 $c > 0$ 。可以再次选择足够大的 c ,使猜测对于基本情况成立,因此,只考虑 $n \geq 2$ 的情

况。如果所做的猜测是，归纳假设对于输入大小小于 n 成立，那么可得

$$\begin{aligned} T(n) &= 2T(n/2) + bn \log n \\ &\leq 2(c(n/2) \log^2(n/2)) + bn \log n \\ &= cn(\log^2 n - 2\log n + 1) + bn \log n \\ &= cn \log^2 n - 2cn \log n + cn + bn \log n \\ &\leq cn \log^2 n \end{aligned}$$

266 只要 $c \geq b$ ，上式即成立。因此，经证明在这种情况下 $T(n)$ 确实是 $O(n \log^2 n)$ 。

在利用这个方法的过程中，必须仔细。如果恰好 $T(n)$ 的一个归纳假设不成立，这并不意味着 $T(n)$ 的另一个归纳假设不成立。

示例5.3 考虑如下递归方程（假设基本情况为 $T(n) = b$ ，其中 $n < 2$ ）：

$$T(n) = 2T(n/2) + \log n$$

这个方程是2.4.4节讨论的自底向上构造堆的运行时间，已证明它的运行时间为 $O(n)$ 。然而，如果试图用最直接的归纳假设证明这个事实，就会碰到一些困难。特别地，考虑如下猜测：

$$\text{首次猜测： } T(n) \leq cn$$

其中常数 $c > 0$ 。可以选择足够大的 c ，使猜测对于基本情况成立，因此，只考虑 $n \geq 2$ 的情况。如果首先所做的猜测是，归纳假设对于输入大小小于 n 成立，那么可得

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2(c(n/2)) + \log n \\ &= cn + \log n \end{aligned}$$

但是，对于 $n \geq 2$ ，不存在任何方法使得上式最后一行小于或等于 cn 。因此，第一个猜测是不充分的。再试

$$\text{更好的猜测： } T(n) \leq c(n - \log n)$$

其中常数 $c > 0$ 。可以再次选择足够大的 c ，使猜测对于基本情况成立，事实上，可以证明它对于 $n < 8$ 成立。因此，只考虑 $n \geq 8$ 的情况。如果所做的猜测是，归纳假设对于输入大小小于 n 成立，那么可得

$$\begin{aligned} T(n) &= 2T(n/2) + \log n \\ &\leq 2c((n/2) - \log(n/2)) + \log n \\ &= cn - 2c \log n + 2c + \log n \\ &= c(n - \log n) - c \log n + 2c + \log n \\ &\leq c(n - \log n) \end{aligned}$$

只要 $c \geq 3$ 且 $n \geq 8$ ，上式即成立。因此，经证明在这种情况下 $T(n)$ 确实是 $O(n)$ 。

267 猜测-测试法可用于建立递归方程渐近复杂度的上界或下界。即便如此，正如上述例子中所示范的，这种方法要求具有一些数学归纳法的技能。

4. 主方法

上述求解递归方程的每个方法都是一个ad hoc，需要利用数学假设才能有效。然而，还有一种求解分治递归方程的方法，具有相当的普适性。这种方法不需要显式正确利用归纳法，称之为主方法（master method）。主方法是表征广泛的递归方程渐近特性的“通用”方法。即它用于如下形式的递归方程：

$$T(n) = \begin{cases} c & \text{如果 } n < d \\ aT(n/b) + f(n) & \text{如果 } n \geq d \end{cases}$$

其中 d 是一个大于或等于1的整型常数, $a > 0$, $c > 0$ 和 $b > 1$ 是实常数。对于 $n \geq d$, $f(n)$ 是正函数。在分析分治递归算法时, 分治法将问题分解为 a 个子问题, 每个子问题规模至多为 n/b , 递归求解每个子问题, 然后将子问题的解“归并”成为整个问题的解, 从而引出上述的递归方程。方程中的函数 $f(n)$ 表示把问题划分为子问题并把子问题的解归并为原问题的解所需的总时间。上面给出的每个递归方程所利用的这种形式与早先本书中给出的利用递归方程分析分治算法得出的形式一样。因此, 这确实是分治递归方程的一般形式。

求解这样的递归方程的主方法根据问题所属的三种情况之一可以得出方程的解。通过把 $f(n)$ 与特殊函数 $n^{\log_b a}$ 进行比较, 区别到底属于哪一种情况(后面将会表明为什么这个特殊函数是如此重要)。

定理5.3[主定理] 设 $f(n)$ 和 $T(n)$ 定义如上。

- (1) 如果存在某个小常数 $\varepsilon > 0$, 满足 $f(n)$ 是 $O(n^{\log_b a - \varepsilon})$, 那么 $T(n)$ 是 $\Theta(n^{\log_b a})$ 。
- (2) 如果存在常数 $k \geq 0$, 满足 $f(n)$ 是 $\Theta(n^{\log_b a} \log^k n)$, 那么 $T(n)$ 是 $\Theta(n^{\log_b a} \log^{k+1} n)$ 。
- (3) 如果存在某些小常数 $\varepsilon > 0$ 和 $\delta < 1$, 满足 $f(n)$ 是 $\Omega(n^{\log_b a + \varepsilon})$, 且对 $n \geq d$ 有 $af(n/b) \leq \delta f(n)$, 那么 $T(n)$ 是 $\Theta(f(n))$ 。

情况(1)表征了 $f(n)$ 多项式级小于特殊函数 $n^{\log_b a}$ 的情况。情况(2)表征了 $f(n)$ 渐近逼近特殊函数 $n^{\log_b a}$ 的情况。情况(3)表征了 $f(n)$ 多项式级大于特殊函数 $n^{\log_b a}$ 的情况。

268

通过几个例子说明了主方法的应用(每个例子都假设 $T(n) = c$, 其中 $n < d$, 常数 $c \geq 1$ 且常数 $d \geq 1$)。

示例5.4 考虑递归方程

$$T(n) = 4T(n/2) + n$$

在这种情况下, $n^{\log_b a} = n^{\log_2 4} = n^2$ 。因此, 这是情况(1), 取 $\varepsilon = 1$, $f(n)$ 是 $O(n^{2-\varepsilon})$ 。这意味着根据主方法可得 $T(n)$ 是 $\Theta(n^2)$ 。

示例5.5 考虑递归方程

$$T(n) = 2T(n/2) + n \log n$$

这是上面给出的递归方程之一。在这种情况下, $n^{\log_b a} = n^{\log_2 2} = n$ 。因此, 这是情况(2), 取 $k = 1$, $f(n)$ 是 $\Theta(n \log n)$ 。这意味着根据主方法可得 $T(n)$ 是 $\Theta(n \log^2 n)$ 。

示例5.6 考虑递归方程

$$T(n) = T(n/3) + n$$

这是初始为 n 的几何递减求和级数的递归方程。在这种情况下, $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$ 。因此, 这是情况(3), 取 $\varepsilon = 1$, $f(n)$ 是 $\Omega(n^{0+\varepsilon})$, 且 $af(n/b) = n/3 = (1/3)f(n)$ 。这意味着根据主方法可得 $T(n)$ 是 $\Theta(n)$ 。

示例5.7 考虑递归方程

$$T(n) = 9T(n/3) + n^{2.5}$$

在这种情况下, $n^{\log_b a} = n^{\log_3 9} = n^2$ 。因此, 这是情况(3), 由于 $f(n)$ 为 $\Omega(n^{2+\varepsilon})$ (因为 $\varepsilon = 1/2$) 且 $af(n/b) = 9(n/3)^{2.5} = (1/3)^{1/2}f(n)$ 。这意味着根据主方法可得 $T(n)$ 为 $\Theta(n^{2.5})$ 。

示例5.8 最后, 考虑递归方程

$$T(n) = 2T(n^{1/2}) + \log n$$

不幸的是, 这个方程不属于可以应用主方法的任何一种情况。但是, 可以引入变量 $k = \log n$, 将方程变为

$$T(n) = T(2^k) = 2T(2^{k/2}) + k$$

将 $S(k) = T(2^k)$ 代入这个方程, 可得

$$S(k) = 2S(k/2) + k$$

现在可以利用主方法, 解得 $S(k)$ 为 $O(k \log k)$ 。代回 $T(n)$, 可得 $T(n)$ 是 $O(\log n \log \log n)$ 。

我们没有严格证明定理5.3, 而是在较高层次上讨论主方法背后的证明过程。

如果将迭代代换法应用到一般分治递归方程上, 可得

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ &= a(aT(n/b^2) + f(n/b)) + f(n) = a^2T(n/b^2) + af(n/b) + f(n) \\ &= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\ &\vdots \\ &= a^{\log_b n}T(1) + \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \\ &= n^{\log_b a}T(1) + \sum_{i=0}^{\log_b n-1} a^i f(n/b^i) \end{aligned}$$

最后一次代换基于恒等式 $a^{\log_b n} = n^{\log_b a}$ 。实际上, 这个方程即为特殊函数的来源。给定 $T(n)$ 的这种封闭形式的表征, 可以直观地看到这三种情况是如何导出的。当 $f(n)$ 较小, 且上述方程中的第一项起着支配作用时, 为情况(1)。当上述求和中的每一项与其他项成比例时, $T(n)$ 的表征为 $f(n)$ 的一个对数因子的倍数, 为情况(2)。最后, 当第一项小于第二项, 且上式求和是 $f(n)$ 作为起始项的一个递减几何级数项的求和时, 因此 $T(n)$ 自身与 $f(n)$ 成正比, 为情况(3)。

定理5.3的证明对这种直觉做了形式化阐述, 但没有给出详细的证明过程。以下给出两个应用主方法的例子。

5.2.2 整数相乘

在这一小节里, 考虑大整数 (big integer) 相乘的问题。大整数就是用大量的位表示的整数, 单个处理器的算术单元不能直接处理这些整数。大整数相乘在数据安全中有重要应用, 它可用于加密模式中。

给定两个用 n 位表示的整数 I 和 J , 很容易用 $O(n)$ 时间计算出 $I+J$ 和 $I-J$ 。但是, 利用通用升级算法有效计算 $I \cdot J$ 的乘积, 所需时间为 $O(n^2)$ 。在本节的其余内容里, 将证明利用分治技术, 可以设计出一个计算两个 n 位整数相乘的亚二次时间的算法。

假设 n 是 2 的幂 (如果不是这种情况, 将给 I 和 J 填充 0)。可以将 I 和 J 的位表示分成两半, 一半表示高位 (higher-order), 另一半表示低位 (lower-order)。特别地, 如果把 I 分成 I_h 和 I_l , 把 J 分成 J_h 和 J_l , 则有

$$\begin{aligned} I &= I_h 2^{n/2} + I_l \\ J &= J_h 2^{n/2} + J_l \end{aligned}$$

270

同时, 观察可知: 二进制数 I 乘以2的幂即 2^k 是简单的, 只需要把数 I 左移(即在高位方向上) k 位。因此, 假如左移操作为常数时间, 则整数与 2^k 相乘所需时间为 $O(k)$ 。

现在把注意力放在计算 $I \cdot J$ 乘积的问题上。给定上述计算 $I \cdot J$ 的展开式, 重写 $I \cdot J$ 如下:

$$I \cdot J = (I_h 2^{n/2} + I_l) \cdot (J_h 2^{n/2} + J_l) = I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

因此, 可以利用分治法计算 $I \cdot J$ 。它把 I 和 J 的位表示一分为二, 递归计算4个位数均为 $n/2$ 的乘积(如上所述), 然后, 利用加法和移位操作, 在 $O(n)$ 时间内归并这些子问题的解。当得到两个1位数的乘法时(这是简单的), 递归过程终止。分治算法的运行时间可以表征为如下递归方程(对于 $n \geq 2$):

$$T(n) = 4T(n/2) + cn$$

其中常数 $c > 0$ 。然后可以应用主定理, 注意在这种情况下, 特殊函数 $n^{\log_b a} = n^{\log_2 4} = n^2$; 因此, 这是情况(1), $T(n)$ 是 $\Theta(n^2)$ 。不幸的是, 这个算法并不比升级算法好。

从主方法可以得到如何改进这个算法的一些见解。如果能够减少递归调用次数, 就会降低主定理中所用的特殊函数的复杂度, 而这个函数当前在运行时间中起着支配作用。幸运的是, 如果我们更加明智地定义递归求解的子问题, 就能把递归调用次数减少1次。特别考虑如下乘积

$$(I_h - I_l) \cdot (J_l - J_h) = I_h J_l - I_l J_l - I_h J_h + I_l J_h$$

这肯定是一个所考虑的奇怪乘积, 但它具有有趣的性质。当对它进行展开时, 其中包含我们想要计算的两个乘积(即 $I_h J_l$ 和 $I_l J_h$)和两个可以递归地计算的乘积(即 $I_h J_h$ 和 $I_l J_l$)。因此, 可以计算 $I \cdot J$:

$$I \cdot J = I_h J_h 2^n + [(I_h - I_l) \cdot (J_l - J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l$$

这个计算要求计算三个位数均为 $n/2$ 的乘积, 加上 $O(n)$ 的额外工作。因此, 可用如下递归方程表征分治算法的运行时间(其中 $n \geq 2$)。

$$T(n) = 3T(n/2) + cn$$

其中常数 $c > 0$ 。

定理5.4 两个 n 位的整数相乘的运行时间为 $O(n^{1.585})$ 。

证明 应用主定理, 在这种情况下, 特殊函数 $n^{\log_b a} = n^{\log_2 3}$; 因此, 符合主定理情况(1), $T(n)$ 为 $\Theta(n^{\log_2 3})$, 即 $O(n^{1.585})$ 。 ■

271

利用分治法, 设计整数相乘的分治算法, 这个算法渐近快于直观的二次时间法。事实上, 如果利用更复杂的分治技术, 还能做得更好, 此时运行时间“几乎”为 $O(n \log n)$ 。这种分治技术称为快速傅里叶变换(fast Fourier transform), 我们将在10.4节讨论它。

5.2.3 矩阵相乘

给定两个 $n \times n$ 的矩阵 X 和 Y , 想要计算它们的乘积 $Z = XY$, 定义如下:

$$Z[i, j] = \sum_{k=0}^{n-1} X[i, k] \cdot Y[k, j]$$

这个方程引出了一个简单的运行时间为 $O(n^3)$ 的算法。

还可把这个乘积看成是子矩阵的乘积, 即假设 n 是2的幂, 把 X 、 Y 和 Z 分别划分成4个大小为 $(n/2) \times (n/2)$ 的矩阵, 重写 $Z = XY$, 得

$$\begin{pmatrix} I & J \\ K & L \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

因此,

$$I = AE + BG$$

$$J = AF + BH$$

$$K = CE + DG$$

$$L = CF + DH$$

在计算 $Z = XY$ 的分治算法中, 可以利用子数组 $A \sim G$, 通过计算 I 、 J 、 K 和 L , 来使用这个方程集。通过上述方程, 由8个递归计算的大小为 $(n/2) \times (n/2)$ 子矩阵乘积, 可计算 I 、 J 、 K 和 L , 再加上运行时间为 $O(n^2)$ 的4个加法操作。因此, 可用如下递归方程表征上述分治算法引出的方程集的运行时间 $T(n)$

$$T(n) = 8T(n/2) + bn^2$$

其中常数 $b > 0$ 。不幸的是, 由主定理可知: 这个方程的运行时间 $T(n)$ 为 $O(n^3)$; 因此, 它并不比直观的矩阵相乘算法好。

有趣的是, 存在一个称为Strassen算法 (Strassen algorithm) 的算法, 它对 $A \sim G$ 子矩阵的计算进行有效组织, 使得只利用7次递归矩阵相乘即可计算 I 、 J 、 K 和 L 的值。Strassen如何发现这些方程仍然是个谜, 但我们只需验证算法是否正确。

272

定义7个子矩阵乘积, 开始Strassen算法。

$$\begin{aligned} S_1 &= A(F - H) \\ S_2 &= (A + B)H \\ S_3 &= (C + D)E \\ S_4 &= D(G - E) \\ S_5 &= (A + D)(E + H) \\ S_6 &= (B - D)(G + H) \\ S_7 &= (A - C)(E + F) \end{aligned}$$

给定这7个子矩阵乘积, 可以计算 I

$$\begin{aligned} I &= S_5 + S_6 + S_4 - S_2 \\ &= (A + D)(E + H) + (B - D)(G + H) + D(G - E) - (A + B)H \\ &= AE + DE + AH + DH + BG - DG + BH - DH + DG - DE - AH - BH \\ &= AE + BG \end{aligned}$$

可以计算 J

$$\begin{aligned} J &= S_1 + S_2 \\ &= A(F - H) + (A + B)H \\ &= AF - AH + AH + BH \\ &= AF + BH \end{aligned}$$

可以计算 K

$$\begin{aligned}
 K &= S_3 + S_4 \\
 &= (C + D)E + D(G - E) \\
 &= CE + DE + DG - DE \\
 &= CE + DG
 \end{aligned}$$

最后计算 L

$$\begin{aligned}
 L &= S_1 - S_7 - S_3 + S_5 \\
 &= A(F - H) - (A - C)(E + F) - (C + D)E + (A + D)(E + H) \\
 &= AF - AH - AE + CE - AF + CF - CE - DE + AE + DE + AH + DH \\
 &= CF + DH
 \end{aligned}$$

因此, 可以利用7个大小为 $(n/2) \times (n/2)$ 的递归矩阵乘积, 计算 $Z = XY$ 。因此表征 $T(n)$ 运行时间的递归方程为

$$T(n) = 7T(n/2) + bn^2$$

其中常数 $b > 0$ 。由主定理可知如下定理:

定理5.5 两个 $n \times n$ 的矩阵相乘的运行时间为 $O(n^{\log 7})$ 。

因此, 只需适量的额外乘法, 计算两个 $n \times n$ 矩阵乘法的运行时间为 $O(n^{2.808})$, 它是 $o(n^3)$ 。实际上, 有许多更复杂的矩阵相乘算法, 其复杂度与Strassen算法相当, 运行时间则少至 $O(n^{2.376})$ 。

273

5.3 动态规划

在这一节里, 讨论动态规划 (dynamic programming) 算法设计技术。这项技术与分治法类似, 可用于求解广泛不同的问题。但是, 概念上讲, 动态规划技术不同于分治法, 因为分治技术可以容易地用一两句话解释它的思想, 且可用单个例子加以说明。而对于动态规划技术, 则需要更多的解释和多个例子说明, 才能完全理解它。

需要更多的努力才能完全理解动态规划方法, 但是值得这样做。有几种算法设计技术对要解决的问题似乎具有指数时间, 却得到求解它们的多项式时间算法。动态规划就是这样一种技术。此外, 应用动态规划技术导致的算法通常相当简单, 代码简练, 只有若干层嵌套循环, 用表数据结构记录循环产生的结果。

5.3.1 矩阵链乘

我们不是从解释一般动态规划技术的原理开始, 而是首先给出一个经典且具体的例子。给定 n 个二维矩阵集合, 希望计算矩阵乘积

$$A = A_0 \cdot A_1 \cdot A_2 \cdots A_{n-1}$$

其中 A_i 是 $d_i \times d_{i+1}$ 矩阵, $i = 0, 1, 2, \dots, n-1$ 。在标准矩阵相乘算法中 (将会用到这个算法), 将一个 $d \times e$ 矩阵 B 与一个 $e \times f$ 矩阵 C 相乘, 计算乘积中的第 (i, j) 个元素

$$\sum_{k=0}^{e-1} B[i, k] \cdot C[k, j]$$

这个定义表明矩阵相乘是可结合的, 即 $B \cdot (C \cdot D) = (B \cdot C) \cdot D$ 。因此, 可以按照想要的方式对 A 的表达式加上括号, 其结果都相同。但是, 每次加上括号之后, 相应执行的基本 (即标量) 乘法次数未必相同。下面举例说明。

示例5.9 设 B 是一个 2×10 矩阵, C 是一个 10×50 矩阵, D 是一个 50×20 的矩阵。计算 $B \cdot (C \cdot D)$ 需要 $2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10\,400$ 次乘法, 而计算 $(B \cdot C) \cdot D$ 需要 $2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3\,000$ 次乘法。

[274]

矩阵链乘问题是要确定 A 的表达式的加括号方式, 使得所执行的标量乘法次数最少。当然, 求解这个问题的一种方法是简单地枚举 A 的表达式的的所有加括号方式, 并确定对于每种加括号方式所执行的乘法次数。不幸的是, A 的表达式的的所有加括号方式集等于有 n 个外部结点的所有不同二叉树集的数量。这个数是 n 的指数。因此, 直接(“蛮力”)算法的运行时间为指数时间, 因为对于一个可结合的算术表达式加上括号存在指数种方式(这个数等于第 n 个Catalan数, 为 $\Omega(4^n/n^{3/2})$)。

1. 定义子问题

可以对蛮力方法所达到的性能进行实质性的改进。因此, 对于矩阵链乘问题的本质, 需作几点思考。第一点思考是可将问题分解为子问题(subproblem)。在这种情况下, 可以定义许多不同的子问题, 每个均用于计算子表达式 $A_i \cdot A_{i+1} \cdots A_j$ 的最佳加括号方式。为简化表示, 令 N_{ij} 表示计算这个子表达式所需的最少乘法次数。因此, 原矩阵的链乘问题可以表征为计算 $N_{0, n-1}$ 的值。这个观察是重要的, 但为了应用动态规划技术, 还需要做进一步的观察。

2. 表征最优解

对于矩阵链乘问题所做的另一个重要观察是, 某个子问题的最优解可以根据它的子问题的最优解定义。称这个性质为子问题最优性(subproblem optimality)条件。

对于矩阵链乘问题, 观察可知无论对子表达式如何加括号, 最终必定需执行某个矩阵的乘法, 即子表达式 $A_i \cdot A_{i+1} \cdots A_j$ 的完全加括号方式一定具有形式 $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$, 其中 $k \in \{i, i+1, \dots, j-1\}$ 。此外, 只要所选的 k 值正确, 一定使得 $(A_i \cdots A_k)$ 和 $(A_{k+1} \cdots A_j)$ 均最优求解。否则, 必定存在全局最优解, 它会对这些子问题之一最优地求解。但这是不可能的, 因为可以用子问题的最优解代替当前子问题的解, 从而减少总乘法次数。这点观察蕴涵着可以根据其他子问题的最优解定义问题 N_{ij} 的最优解, 即考虑 k 的每个取值, 并取使乘法次数最少的那个 k 值。

[275]

3. 设计一个动态规划算法

上述讨论蕴涵着可以将子问题 N_{ij} 的最优解表征为

$$N_{ij} = \min_{i \leq k < j} \{N_{ik} + N_{k+1j} + d_i d_{k+1} d_{j+1}\}$$

其中

$$N_{ii} = 0$$

因为对于包含一个矩阵的子表达式不需执行任何计算。即对于 k 的每个值, 计算相应的每个子表达式所需的乘法次数, 再加上进行最终矩阵相乘所需的乘法次数, 选择使 N_{ij} 达到最小值的 k 值。

N_{ij} 的方程类似于分治算法导出的递归方程, 但这只是在表面上类似。因为存在方程的 N_{ij} 某个方面, 使得难以利用分治法计算 N_{ij} 。这就是存在子问题不断共享的问题, 使我们不能把问题划分成完全独立的子问题(就像应用分治技术所做的那样)。然而, 可以按照自底向上的方式, 通过计算 N_{ij} 的值利用 N_{ij} 的方程导出有效算法, 并把所得的中间解存储在由 N_{ij} 值组成的表中。首先足够简单地设 $N_{ii} = 0$, 其中 $i = 0, 1, \dots, n-1$ 。然后, 利用方程 N_{ij} 的一般形式计算 $N_{i,i+1}$ 的值, 因为它们只依赖于 N_{ii} 和 $N_{i+1,i+1}$ 的值, 它们是可用的。给定 $N_{i,i+1}$ 的值, 就可以计算 $N_{i,i+2}$ 的值, 依此类推。于是, 可由先前计算得到的值计算 N_{ij} , 直到最终计算 $N_{0,n-1}$ 的值。这个值正是我们所想要的。算法5-3给出了这个动态规划求解的细节。

算法5-3 矩阵链乘问题的动态规划算法

```

算法 MatrixChain( $d_0, \dots, d_n$ ):
  输入: 整数序列  $d_0, \dots, d_n$ 
  输出: 对于  $i, j = 0, \dots, n-1$ , 计算乘积  $A_i A_{i+1} \dots A_j$  所需的最少乘法次数  $N_{i,j}$ , 其中  $A_k$  是  $d_k \times d_{k+1}$  矩阵
  for  $i \leftarrow 0$  to  $n-1$  do
     $N_{i,i} \leftarrow 0$ 
  for  $b \leftarrow 1$  to  $n-1$  do
    for  $i \leftarrow 0$  to  $n-b-1$  do
       $j \leftarrow i + b$ 
       $N_{i,j} \leftarrow +\infty$ 
      for  $k \leftarrow i$  to  $j-1$  do
         $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$ 

```

276

4. 矩阵链乘算法分析

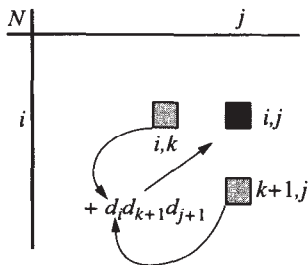
因此, 可以通过主要由3层嵌套for循环组成的算法计算 $N_{0,n-1}$ 的值。外层循环执行 n 次。内层循环至多执行 n 次。最内层的循环也至多执行 n 次。于是, 算法的总运行时间为 $O(n^3)$ 。

定理5.6 给定 n 个二维矩阵的链乘, 可以在 $O(n^3)$ 时间内计算这个链乘的加括号方式, 使标量乘法的次数最少。

证明 上面已经证明如何计算标量乘法的最优次数。但如何恢复实际的括号呢?

计算括号位置自身的方法实际上是相当直观的。可以修改计算 $N_{i,j}$ 值的算法, 使得每当找到 $N_{i,j}$ 的一个新的最小值时, 则将使 $N_{i,j}$ 达到最小值的下标 k 存储起来。■

在图5-3中, 说明了矩阵链乘问题的动态规划解法填充数组 N 的方式。

图5-3 矩阵链乘问题的动态规划算法填充数组 N 的方式的说明

既然已经了解了使用动态规划方法的一个完整的示例, 现在可以讨论将动态规划技术应用于求解其他问题的一般方面。

277

5.3.2 一般技术

动态规划技术主要用于求解优化 (optimization) 问题, 在优化问题中希望找出求解问题的“最佳”方式。这些求解的方式常常是指数级的。除规模较小的问题之外, 蛮力搜索寻求最佳的方式在计算上是不可行的。但是, 在这些情况下, 只要问题具有能够探索的某种结构, 就可以利用动态规划技术求解。这种结构包括以下三个部分:

简单的子问题: 必定有一种方式把全局优化问题分解成子问题, 每个子问题在结构上与原问

题类似。此外，还要有一种简单方式，只利用几个下标（如 i 、 j 和 k 等）就可以定义子问题。

子问题的最优性：利用相对简单的组合操作，原问题的最优解必定包括子问题的最优解。如果包括子问题的次优解，则不能得到原问题的最优解。

子问题的重叠性：无关子问题的最优解可以包括公共子问题。这样的重叠提高了动态规划算法存储子问题解的效率。

既然已经给出了动态规划算法的一般组成部分，下面给出其应用的另一个例子。

5.3.3 0-1 背包问题

假定一个徒步旅行者准备艰苦跋涉，带着一个背包穿过雨林。进一步假定她知道背包所能承受的总权值 W ，以及她想要携带的 n 个不同的有用物品集 S ，如折叠椅、帐篷、本书的副本。假设每个物品 i 有一个整数权值为 w_i ，一个效用值 b_i ，这是旅行者赋予物品 i 的效用值。她的问题是在不超出权值限制 W 的前提下，优化所能携带的物品集 T 的总价值。即她的目标函数为

$$\text{maximize } \sum_{i \in T} b_i, \text{ 约束条件为 } \sum_{i \in T} w_i \leq W$$

278 她的问题是0-1背包问题（0-1 knapsack problem）的一个实例。这个问题之所以称为“0-1”问题，是因为每个物品要么完全接受，要么完全拒绝。在5.1.1节中，考虑了背包问题的部分物品版本。在习题R-5.12中，研究了在因特网拍卖的环境中是如何引出背包问题的。

1. 首次尝试表征子问题

可以很容易在 $\Theta(2^n)$ 时间内解决0-1背包问题，其方式是：枚举 S 的所有子集，并选择那个具有最高效用且不超过背包权重 W 的子集。但是这是一个效率低下的算法。幸运的是，可以导出0-1背包问题的动态规划算法，在大多数情况下，这个动态规划算法比上述算法快得多。

像许多动态规划问题一样，为0-1背包问题设计这样一个算法最困难的部分之一在于找出子问题的良好表征（使得满足一个动态规划算法的三个性质）。为了简化讨论，将 S 中的物品编号为 $1, 2, \dots, n$ ，对于每个 $k \in \{1, 2, \dots, n\}$ ，定义子集

$$S_k = \{S \text{ 中的物品编号为 } 1, 2, \dots, k\}$$

定义子问题的一种可能的方法是利用参数 k ，满足只用 S_k 中的物品，子问题 k 是充填背包的最佳方式。这是一个有效的子问题定义，但是根本不清楚如何根据子问题的最优解，定义下标为 k 的子问题的最优解。我们希望利用 S_{k-1} 中的物品，并考虑如何将物品 k 添加到 S_{k-1} 中，导出可以获得最优解的方程。不幸的是，如果坚持这个子问题的定义，那么，这种方法具有致命的弱点。这是因为，正如图5-4所示，如果利用它来表征子问题，那么原问题的最优解实际上可能包含子问题的次优解。

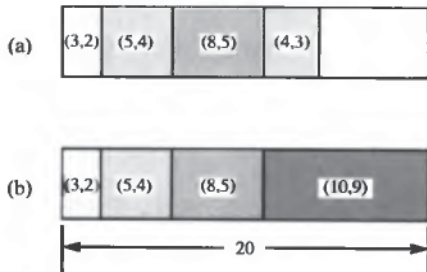


图5-4 示例显示定义背包子问题的第一种方法并不奏效。集 S 由五个用(权值, 效用)对表示的物品组成，它们是(3, 2)、(5, 4)、(8, 5)、(4, 3)和(10, 9)。背包总权值 $W=20$ ：(a)前四个物品的最优解；(b)前五个物品的最优解。每个物品的阴影度与其效用成正比

2. 更好地表征子问题

仅根据下标 k 定义子问题的致命弱点是, 子问题的表示中没有提供足够的信息以帮助求解优化问题。可以通过增加一个参数 w , 修正这个难点。因此, 把每个子问题的计算形式化为 $B[k, w]$, 它被定义为 S_k 中一个子集的最大效用值, 它是 S_k 的所有子集中权值至多为 w 的子集。对于每个 $w \leq W$, 定义 $B[0, w] = 0$ 。我们导出以下一般情况下的关系

$$B[k, w] = \begin{cases} B[k-1, w] & \text{如果 } w_k > w \\ \max\{B[k-1, w], B[k-1, w-w_k] + b_k\} & \text{其他情况} \end{cases}$$

即权值至多为 w 的 S_k 的最优子集, 它要么是权值至多为 w 的 S_{k-1} 的最优子集, 要么是权值至多为 $w-w_k$ 的 S_{k-1} 的最优子集, 再加上物品 k 。因为权值至多为 w 的 S_k 最优子集要么包含物品 k , 要么不包含物品 k 。其中之一必是正确选择。因此, 定义了一个具有两个参数 k 和 w 的简单子问题, 且满足子问题最优性条件。然而, 还有子问题重叠问题, 因为权值至多为 w 的最优子集会在将来的许多子问题中被使用。

在从此定义导出算法的过程中, 可以进行另外一些观察, 即通过 $B[k-1, w]$ 构造 $B[k, w]$, 可能还有 $B[k-1, w-w_k]$ 。因此, 只利用一个数组 B 就可实现这个算法, 在以参数 k 为下标的每次迭代中, 更新 B 的值。在每次迭代的最后, $B[w] = B[k, w]$ 。如算法5-4 (01Knapsack) 所示。

算法5-4 求解0-1背包问题的动态规划算法

算法 01Knapsack(S, W):

输入: n 个物品集, 满足物品 i 有正效用 b_i 和正整型权值 w_i ; 正最大整型总权值 W

输出: 对于 $w = 0, \dots, W$, 在总权值至多为 w 的条件下, 使 S 的子集的效用 $B[w]$ 达到最大

for $w \leftarrow 0$ to W do

$B[w] \leftarrow 0$

for $k \leftarrow 1$ to n do

 for $w \leftarrow W$ downto w_k do

 if $B[w-w_k] + b_k > B[w]$ then

$B[w] \leftarrow B[w-w_k] + b_k$

280

3. 0-1背包问题的动态规划算法分析

01Knapsack算法的运行时间由两层嵌套的for循环支配, 外层循环迭代 n 次, 内层循环至多迭代 W 次。通过在所有 $w \leq W$ 中找出最大的 $B[w]$ 值, 得到问题的最优解。由此可得:

定理5.7 给定整数 W 和 n 个物品的集合 S , 每个物品有正效用和正整型权值, 可以用 $O(nW)$ 时间求出 S 中总权值至多为 W 且具有最大效用的子集。

证明 算法5-4 (01Knapsack) 利用数组 B 保存最佳子集, 在总权值至多为 W 的条件下, 构造 S 的一个最大效用子集的值。可以很容易地把算法转换成一个输出最优子集中物品的算法, 但是把这个转换的细节留作习题。 ■

4. 伪多项式时间算法

除了作为动态规划的另一个应用之外, 定理5.7还指出了某些有趣的性质, 即指出算法的运行时间依赖于参数 W , 严格地讲, 算法的运行时间并不与输入大小 (n 个物品及其权值和效用, 加上数 W) 成正比。假设 W 按照某种标准方式 (如二进制数) 编码, 那么, 只需 $O(\log W)$ 位对 W 编码。此外, 如果 W 很大 (如 $W = 2^n$), 那么这个动态规划算法实际上渐近地比蛮力方法要慢。因

此,从技术上讲,这个算法不是多项式时间算法,因为它的运行时间实际上不是输入大小(size)的函数。

一般称像背包动态规划这样的算法为伪多项式时间(pseudo-polynomial time)算法,因为它的运行时间依赖于输入中给定某个数的量,而不是它的编码大小。实际中,这样的算法应该比蛮力算法快得多,但是说它是真正的多项式时间的算法是不正确的。有一种称为NP完全性(NP-Completeness)的理论,将在第13章进行讨论,它指出极不可能找到0-1背包问题的真正的多项式时间算法。

在本书的其他地方,给出了动态规划技术的其他一些应用,包括计算有向图中的可达性(6.4.2节),以及用于测试两个串的相似性(9.4节)。

281

5.4 习题

基础题

- R-5.1 设 $S = \{a, b, c, d, e, f, g\}$ 是对象的集合,每个对象具有效用-权值对,如下: $a: (12, 4)$ 、 $b: (10, 6)$ 、 $c: (8, 5)$ 、 $d: (11, 7)$ 、 $e: (14, 3)$ 、 $f: (7, 1)$ 、 $g: (9, 6)$ 。假设背包总权值为18,求 S 的部分背包问题的最优解。说明你的工作。
- R-5.2 描述如何用 $O(n \log n)$ 时间实现TaskSchedule方法。
- R-5.3 给定由开始时间和完成时间对组成的任务集 $T = \{(1, 2), (1, 3), (1, 4), (2, 5), (3, 7), (4, 9), (5, 6), (6, 8), (7, 9)\}$ 。求解此任务集的任务调度问题。
- R-5.4 利用主方法,表征以下递归方程(假设对于 $n < d$, $T(n) = c$, 其中常数 $c > 0$ 且 $d \geq 1$)。
- $T(n) = 2T(n/2) + \log n$
 - $T(n) = 8T(n/2) + n^2$
 - $T(n) = 16T(n/2) + (n \log n)^4$
 - $T(n) = 7T(n/3) + n$
 - $T(n) = 9T(n/3) + n^3 \log n$
- R-5.5 利用5.2.2节的分治算法,计算二进制表示的10110011-10111010。说明你的工作。
- R-5.6 利用Strassen矩阵相乘算法,计算以下矩阵的乘积。

$$X = \begin{pmatrix} 3 & 2 \\ 4 & 8 \end{pmatrix}, Y = \begin{pmatrix} 1 & 5 \\ 9 & 6 \end{pmatrix}$$

- R-5.7 复数 $a + bi$ (其中 $i = \sqrt{-1}$)可用数对 (a, b) 表示。描述一个方法,只进行三次实数乘法,计算表示 $a + bi$ 和 $c + di$ 相乘的数对 (e, f) 。
- R-5.8 布尔矩阵就是那些元素值为0或1的矩阵,利用AND表示,利用OR表示+,进行矩阵相乘。给定两个 $n \times n$ 的随机布尔矩阵 A 和 B ,满足任何元素值均为1的概率为 $1/k$ 。证明如果 k 是常数,那么存在一个执行矩阵 A 和 B 相乘且运行时间为 $O(n^2)$ 的算法。如果 k 值为 n ,结果如何?
- R-5.9 给定矩阵的维数 10×5 、 5×2 、 2×20 、 20×12 、 12×4 和 4×60 ,计算这些矩阵链乘的最佳加括号方式。说明你的工作。
- R-5.10 设计一个矩阵链乘问题的有效算法,输出矩阵相乘的完全加括号表达式。要求使用的操作次数最少。
- R-5.11 对于习题R-5.1,解相应的0-1背包问题。
- R-5.12 Sally正在主持一场因特网拍卖会,出售 n 个物品。她接到 m 次出价,每次出价具有“我想用 d_i 美元购买物品 k_i ”这样的形式,其中 $i = 1, 2, \dots, m$ 。把她的优化问题表征为背包问题。在什么条件下,这个0-1背包问题对应于部分背包问题?

282

创新题

- C-5.1 一位名为Anatjari的土生澳大利亚人想要穿越一片沙漠,他只携带一只水瓶。他有一张地图,标记了沿途经过的所有水洞。假设每行进 k 英里,需要一瓶水。设计一个有效算法,确定Anatjari应该在什么地方充水,使得停止行进的次数尽可能最少。论证你的算法为什么是正确的。
- C-5.2 考虑单机调度(machine scheduling)问题。给定任务集合 T ,以及每项任务的开始时间和完成时间。与在任务调度问题中一样,只不过现在只有一台机器可用,并且目标是最大化在这台机器上执行的任务数。设计这个单机调度问题的一个贪心算法,并证明它是正确的。该算法的运行时间是多少?
- C-5.3 描述钱币兑换问题的一个有效贪心算法。目标是利用最少量的硬币,兑换某个指定的价值。假设有四种硬币(25美分、10美分、5美分和1美分)可用,其价值分别为25、10、5、1。证明你的算法是正确的。
- C-5.4 给出一组硬币面值的示例集合,使得贪心算法并没有利用这个集合的最少硬币数。
- C-5.5 在艺术画廊守卫(art gallery guarding)问题中,给定表示画廊中的长走廊的直线 L 。另外给定实数集合 $X = \{x_0, x_1, \dots, x_{n-1}\}$,表示长廊中油画的位置。假定一个卫兵可以保护与他或她距离至多为1(两边)的所有油画。设计一个找出卫兵布局的算法,利用最少的卫兵数,保证其位置在 X 中的所有油画的安全。
- C-5.6 设计一个分治算法,找出 n 个元素中的最小与最大元素,要求使用的比较次数不超过 $3n/2$ 。
- C-5.7 给定某个运动中 n 个小组的集合 P ,循环赛(round-robin tournament)是一个比赛集合,其中每个小组只与其他小组比赛一次。设计一个有效算法,构造一场循环赛,假设 n 是2的幂。
- C-5.8 给定 $[0, 1]$ 区间上的区间集 $S = \{[a_0, b_0], [a_1, b_1], \dots, [a_{n-1}, b_{n-1}]\}$,且 $0 \leq a_i < b_i \leq 1$,其中 $i = 0, 1, \dots, n-1$ 。进一步假定赋予 S 中的每个区间 $[a_i, b_i]$ 一个高度 h_i 。定义 S 的上包围(upper envelope)为数对组成的表 $[(x_0, c_0), (x_1, c_1), (x_2, c_2), \dots, (x_m, c_m), (x_{m+1}, 0)]$,且 $x_0 = 0, x_{m+1} = 1$,并按照 x_i 的值排序,满足对于每个子区间 $s = [x_i, x_{i+1}]$, S 中包含 s 的最高区间的高度为 c_i ,其中 $i = 0, 1, \dots, m$ 。设计一个计算 S 上包围的 $O(n \log n)$ 时间的算法。
- C-5.9 可以如何修改简单计算0-1背包问题最佳效用值的动态规划算法,使其计算给出这个效用值的分配。
- C-5.10 给定其和为 N 的 n 个正整数的集合 $A = \{a_1, a_2, \dots, a_n\}$ 。设计一个 $O(nN)$ 时间的算法,确定是否存在一个子集 $B \subset A$,满足 $\sum_{a_i \in B} a_i = \sum_{a_i \in A-B} a_i$ 。
- C-5.11 设 P 是一个凸多边形(12.5.1节)。 P 的一个三角剖分(triangulation)是 P 的顶点连接的对角线之和,满足每个内表面是一个三角形。三角剖分的权值是这些对角线长度之和。假设计算长度、加法和比较所需的时间为常数,给出一个计算 P 的最小权值的三角剖分的有效算法。
- C-5.12 语法(grammar) G 是由非终结符号 S 通过称为产生式(production)的简单代换规则产生的“终结”字符串。如果 $B \rightarrow \beta$ 是一个产生式,那么可将一个形如 $\alpha B \gamma$ 的串转换为串 $\alpha \beta \gamma$ 。如果每个产生式具有形如“ $A \rightarrow BC$ ”或“ $A \rightarrow a$ ”的形式,则语法是乔姆斯基范式(chomsky normal form),其中 A, B 和 C 是非终结字符, a 是终结字符。设计一个 $O(n^3)$ 时间的动态规划算法,确定是否可从起始符号 S 产生串 $x = x_0 x_1 \dots x_{n-1}$ 。
- C-5.13 给定 n 个结点的有根树 T ,且 T 中每个结点 v 的权值为 $w(v)$ 。 T 的独立集(independent set)是 T 中结点的一个子集 S ,满足 S 中不存在任何结点是 S 中其他结点的子结点或者父结点。设计一个有效的动态规划算法,找出 T 中结点的最大权值独立集,其中一个结点集的权值就是那个集中结点的权值之和。并分析你的算法的运行时间。

283

程序设计

- P-5.1 设计和实现一个大整数包,支持四种基本算术运算。
- P-5.2 实现有效求解背包问题的一个系统。你的系统应该同时适合于部分背包问题或者0-1背包问题。进行实验分析,测试系统的效率。

5.5 本章注记

术语“贪心算法”是1971年由Edmonds[64]创造的，尽管在那之前这个概念已经存在。关于贪心方法及支持它的理论的更多信息，可参考Papadimitriou和Steiglitz的著作[164]，该理论称为拟阵理论。

分治技术是数据结构和算法设计悠久传统的一部分。求解分治递归方程的主方法源于Bentley、Haken和Saxe的一篇文章[30]。运行时间为 $O(n^{1.585})$ 的两个大整数相乘的分治算法一般归功于俄罗斯人Karatsuba和Ofman[111]。对于两个 n 位数字的数相乘，已知最快的渐近算法是基于FFT的算法，它是Schönhage和Strassen[181]提出的运行时间为 $O(n \log n \log \log n)$ 的算法。

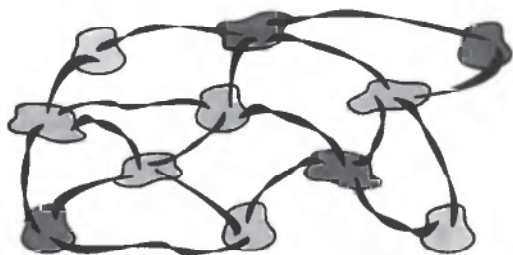
动态规划方法在运筹学团体中得到发展，Bellman[26]对其做了形式化阐述。Godbole[78]描述了矩阵链乘问题的解法。渐近最快的方法由Hu和Shing [101, 102]提出。Hu的著作[100]中描述了背包问题的动态规划算法。与此同时，Hirschberg[95]显示了如何求解最长公共子串问题，但利用线性空间（也可参考[56]）。

图 算 法

第二部分

Part 2





图

据说在希腊神话中，有一个精美的迷宫，里面住着一个一半是公牛、一半是人的怪物 Minotaur。这个迷宫是如此复杂，以至于不论是禽兽还是人类都无法从中逃离。直到希腊英雄忒修斯在国王的女儿 Ariadne 的帮助下，决定实现本章讨论的一个算法。忒修斯在迷宫的门上紧扣了一个带有丝线的球，当他行进在弯弯曲曲的通路中搜寻怪物时，则将门扣放松。显然，忒修斯知道良好的算法设计方法，因为找到并击败怪物之后，忒修斯很容易沿原路出迷宫，返回到所爱的人 Ariadne 的怀抱。

由于能够确定哪一个对象（如迷宫的通路）和其他对象相通，这可能并不总是和在故事中所述的那样重要，但是这是基础性的。例如，在城市地图中存在连通性信息，其中对象是道路。同样，在因特网的路由表中，对象是计算机。连通信息也存在于二叉树所定义的父-子关系中，其中对象是树结点。实际上，连通性信息可以被存在于一对对象之间的各种关系所定义。因此，本章所讨论的主题——图（graph）——关注的是有效处理这些关系的表示和算法，即图是由对象（称为顶点）集合及它们之间的连接集合组成。不应该把“图”的这个概念与条形图和函数绘图相混淆，因为这些种类的“图”与本章讨论的主题无关。

图的应用领域广泛，包括绘图（地理信息系统）、运输（道路网络和航班网络）、电子工程（电路中）和计算机网络（因特网互连中）。因为图的应用是如此广泛和多样，人们给出了大量术语描述图的各个部分和性质。幸运的是，大多数图的应用都是最近的成果，这些术语相当直观。

因此，本章首先回顾图的术语并介绍图 ADT，包括图的一些基本性质。在给出图 ADT 后，6.2 节介绍表示图的三种主要数据结构。与树一样，遍历是图的重要计算，在 6.3 节将会讨论这样的计算。6.4 节讨论有向图，在其中给定关系的方向，连通性问题变得更有意义。最后，在 6.5 节给出一个 Java 实现的深度优先查找的案例研究。这个案例研究中使用了两个软件工程设计模式——修饰模式和模板方法模式，并且讨论了如何将深度优先查找应用于垃圾收集。

288

6.1 图抽象数据类型

抽象地看，图（graph） G 可看作由顶点（vertex）集合 V 和 V 中顶点对构成的边（edge）集合 E 组成。因此，图是表示集合 V 中对象对之间的连接和关系的一种方式。而在一些书籍中，利用图的不同术语，把顶点称为结点（node），边称为弧（arc）。我们利用术语“顶点”和“边”。

图中的边要么是有向的（directed），要么是无向的（undirected）。如果 (u, v) 对是有序的，且 u

是 v 的前驱, 则称从 u 到 v 的边 (u, v) 是有向的。如果 (u, v) 对是无序的, 则称从 u 到 v 的边 (u, v) 是无向的。无向边有时用集合 $\{u, v\}$ 表示, 但为简单起见, 用 (u, v) 表示。注意, 对于无向的情况, (u, v) 与 (v, u) 相同。图中通常把顶点可视化为一圆或矩形, 把边可视化为一线段或曲线, 用于把圆或矩形连接起来。

示例6.1 可以把某一学科研究人员之间的合作关系可视化为一图。图中的顶点是研究员自身, 边表示研究人员之间合著有论文或书籍, 如图6-1所示。这样的边是无向的, 因为合作关系是对称关系 (symmetric relation)。也就是说, 如果 A 与 B 有合作关系, 那么 B 与 A 必定有合作关系。

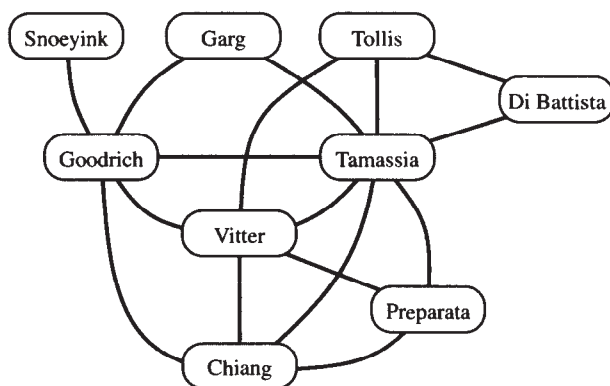


图6-1 某些作者之间的合作关系图

示例6.2 可以把一个面向对象程序关联一个图, 图中的顶点表示程序中定义的类, 边表示类之间的继承性。如果 v 的类扩展了 u 的类, 则存在一条从顶点 v 到 u 的边。这样的边是有向的, 因为继承关系只存在于一个方向上 (即它是非对称的)。

289

如果图中的边是无向的, 那么称图是无向图 (undirected graph)。同样, 有向图 (directed graph) 也记作 digraph, 表示其中的边都是有向的。如果图中的边既是有向的又是无向的, 则称为混合图 (mixed graph)。注意通过用一对有向边 (u, v) 与 (v, u) 代替每一条无向边, 就可把一个无向图或混合图转换成为一个有向图。但是, 保持无向图或混合图如其定义的形式, 常常有许多应用。

示例6.3 可用图将城市地图模型化。图中顶点表示交叉点或终结点, 边表示一段没有交叉点的街道。对于图中的边, 如果其中有两条无向边, 则对应双向道路。有向图则对应单向道路。因此, 模型化城市地图的图是混合图。

示例6.4 图的物理例子出现在一座建筑物的电线和铅锤网络中。可将这样的网络模型化为图, 其中每个连接器、装置或出线盒看作一个顶点, 每条不间断的电线或管道看作一条边。实际上, 这样的图是更大图的一部分, 如本地配电网或配水网。取决于我们对这些图的特定方面的兴趣, 原则上, 水可以在管道中流动, 电流可以在电线的任何一个方向流动。

称由边连接的两个顶点为边的端点 (end vertice)。边的端点也称为边的端点 (endpoint)。如果一条边是有向的, 它的第一个端点是边的源点 (origin), 另一个是边的目的点 (destination)。

如果两个顶点是同一条边的两个端点, 则称这两个顶点是相邻的 (adjacent)。如果一个顶点是边的一个端点, 则这条边依附 (incident) 这个顶点。顶点的出边 (outgoing edge) 就是源点为这个顶点的有向边。顶点的入边 (incoming edge) 就是边的目的点为这个顶点的有向边。顶点 v

的度 (degree) 为依附 v 的边数, 用 $\deg(v)$ 表示。顶点 v 的入度 (in-degree) 和出度 (out-degree) 分别定义为 v 的入边数和出边数, 分别用 $\text{indeg}(v)$ 和 $\text{outdeg}(v)$ 表示。

示例6.5 可将空中交通问题构造为一个图 G , 称为航班网络。顶点关联机场, 边关联航班 (如图6-2所示)。在图 G 中, 边是有向的, 因为给定航班有特定的旅行方向 (从源机场到目的机场)。 G 中一条边 e 的端点分别关联那个航班的源机场和目的机场。如果在 G 的两个机场之间存在航班, 则这两个机场是相邻的。如果 e 的航班飞向 v 的机场或飞出 v 的机场, 则边 e 依附顶点 v 。顶点 v 的出边对应飞出 v 的机场的那些航班, 顶点 v 的入边则对应飞向 v 的机场的那些航班。 G 中顶点 v 的入度对应飞向 v 的机场的航班数, G 中顶点 v 的出度则对应飞出 v 的机场的航班数。

290

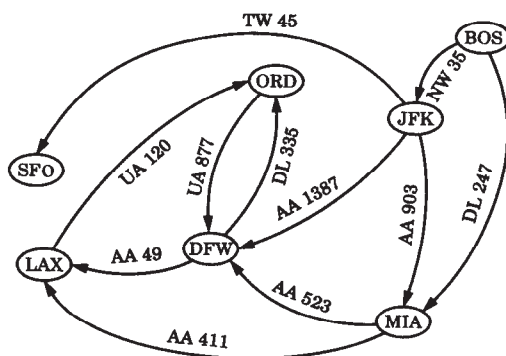


图6-2 表示航班网络的有向图例子。边UA 120的端点是LAX和ORD。因此, LAX和ORD是相邻的。DFW的入度为3, 出度为2

图的定义把边组织在集 (collection) 中, 不是集合 (set) 中, 从而允许两条无向边有相同的端点。对于两条有向边, 则允许有相同的源点和目的点。称这样的边为平行边 (parallel edge) 或者多条边 (multiple edge)。平行边可能存在于航班网络中 (示例6.5)。在这种情况下, 同一对顶点之间的多条边表示那天不同时间同一航线的不同航班。另一种特殊类型的边是顶点连向自己。在这种情况下, 如果一条边的两端点重合, 则称这条边 (无向或有向) 是自环的 (self-loop)。自环可能出现在关联城市地图的一个图中 (示例6.3), 其中它对应一个“圆圈” (回到其起点的曲线大街)。

除了少数几个例外, 像上述提到的那些, 称图中没有平行边或自环的图为简单 (simple) 图。因此, 通常说简单图的边是顶点对的集合 (不是集)。除非特殊说明, 本章都假设图是简单图。这种假设简化了图的数据结构和算法表示。把本章的结论扩展到带有自环或者平行边的一般图, 是直接而又乏味的。

在以下的定理中, 探索图中度和边数的几个重要性质。这些性质把图中顶点数和边数彼此联系起来, 以及把它们与顶点的度数联系起来。

定理6.1 如果 G 是具有 m 条边的图, 那么

$$\sum_{v \in G} \deg(v) = 2m$$

证明 上述求和中, 边 (u, v) 被计算两次; 一次被它的端点 u 计算, 另一次被它的端点 v 计算。

因此, 边对顶点度数的总贡献为边数的两倍。 ■

定理6.2 如果 G 是具有 m 条边的图, 那么

$$\sum_{v \in G} \text{in deg}(v) = \sum_{v \in G} \text{out deg}(v) = m$$

证明 在一个有向图中, 边 (u, v) 对于源点 u 的出度贡献一个单位, 对于目的点 v 的入度贡献一个单位。因此, 边对于顶点出度的总贡献等于边数, 同样, 边对于顶点入度的总贡献也是如此。■

定理6.3 如果 G 是具有 n 个顶点、 m 条边的简单图。如果 G 是无向的, 那么 $m \leq n(n-1)/2$; 如果 G 是有向的, 那么 $m \leq n(n-1)$ 。

证明 假定 G 是无向的。因为 G 中不存在具有相同端点的两条边, 也没有自环。在这种情况下, G 中一个顶点的最大度为 $n-1$ 。因此, 由定理6.1可知, $2m \leq n(n-1)$ 。现假设 G 是有向图。因为 G 中不存在具有相同源点和目的点的两条边, 也没有自环, 在这种情况下, G 中一个顶点的最大入度为 $n-1$ 。因此, 由定理6.2可知, $m \leq n(n-1)$ 。■

换一种方式来看, 定理6.3阐述 n 个顶点的简单图有 $O(n^2)$ 条边。

图中的一条路径 (path) 是一个顶点和边交替组成的序列, 在一个顶点开始, 在一个顶点结束。满足每条边依附它的直接前驱和直接后继顶点。回路 (cycle) 是一条起始顶点和结束顶点相同的路径。如果一条路径中的每个顶点均不同, 则称这条路径是简单的 (simple)。如果除了第一个和最后一个顶点之外, 一个回路中的每个顶点均不同, 则称这条回路是简单的。有向路径 (directed path) 是一条路径, 满足所有边是有向的, 且可沿着其方向遍历。有向回路 (directed cycle) 的定义类似。

示例6.6 给定表示城市地图的有向图 G (参考示例6.3)。有一对夫妇驾车从其家中到某一推荐餐馆用餐, 要行进一条通过 G 的路径, 可以把他们行进的路线模型化为遍历一条路径。如果他们知道路线, 不会意外地经过相同的顶点两次, 那么他们会遍历 G 中的一条简单路径。同样, 可以把这对夫妇从家中到餐馆、再回到家中所行进的路线模型化为一个回路。如果他们从餐馆回家的路线完全不同于他们去往餐馆的路线, 甚至不会经过相同的顶点两次, 那么他们的整个环形旅程就是一个简单的回路。最后, 如果他们沿着单向街道行进整个旅程, 那么可以把他们的夜晚出行模型化为一个有向图。

图 G 的一个子图是图 H , 其顶点和边分别是 G 的顶点和边的子集。 G 的生成子图 (spanning subgraph) 是 G 的子图, 包含图 G 的所有顶点。如果对于图中的任意两个顶点, 其间存在一条路径, 则称图是连通的 (connected)。如果图 G 是不连通的, 称它的最大连通子图为 G 的连通分量 (connected component)。森林 (forest) 是没有回路的图。树 (tree) 是一个连通森林, 即没有回路的连通图。

292

注意这里树的定义稍微不同于2.3节中的定义, 即在图的环境中, 树没有根。定义还是有些模糊, 2.3节的树应该称作有根树 (rooted tree), 而本章的树应该称作自由树 (free tree)。森林的连通分量是 (自由) 树。图的生成树是一个生成子图, 它是一棵 (自由) 树。

示例6.7 今天这个时代谈论最多的也许就是因特网。可将因特网看作一个图, 其顶点是计算机, 其 (无向) 边是因特网上计算机之间的通信连接。计算机和通信连接位于某个域内, 如wiley.com, 构成因特网的一个子图。如果这个子图是连通的, 那么这个域内计算机上的两个用户可以相互发送电子邮件, 而其信息包不需离开这个域。假定这个子图的边构成一棵生成树。这蕴涵着即使单独一条连接发生问题 (例如, 某人把通信电缆从这个域中的计算机上拔出), 那么这个子图不再连通。

树、森林和连通图有许多简单的性质。

定理6.4 设 G 是具有 n 个顶点、 m 条边的无向图。那么 G 有如下性质：

- 如果 G 是连通的，那么 $m \geq n-1$ 。
- 如果 G 是一棵树，那么 $m = n-1$ 。
- 如果 G 是森林，那么 $m \leq n-1$ 。

这个定理的证明留作习题（C-6.1）。

图的方法

作为一种抽象数据类型，图是元素的位置容器，用于存储图中的顶点和边，即图中的位置（position）是它的顶点和边。因此，可以把元素或者边（或者两者）存放在图中。根据面向对象实现方法，这种选择蕴涵着可以定义顶点和边ADT，从而对位置ADT进行了扩展。回忆2.2.2节，一个位置有一个element()方法，它返回存储在这个位置上的元素。我们也可以分别利用顶点和边的特殊迭代器。因为图是一个位置容器，图抽象数据类型支持方法size()、isEmpty()、elements()、positions()、replaceElement(p, o)和swapElements(p, q)，其中 p 和 q 表示位置， o 表示对象（即元素）。

293

与前面章节中讨论过的抽象数据类型相比，图的抽象数据类型要丰富得多。这些丰富的数据类型大多由定义图的两种位置即顶点和边导出。因而介绍图的方法尽可能有组织。我们把图方法分成三类：一般方法、与有向边相关的方法及更新和修改图的方法。此外，为了简化表示，用 v 表示顶点位置， e 表示边位置， o 表示存储顶点或边的对象（元素）。同时，我们不讨论可能出现的错误条件。

1. 一般方法

首先描述图的基本方法，忽略边的方向。以下每个方法返回图 G 的一个全局信息：

- numVertices(): 返回 G 中的顶点数。
- numEdges(): 返回 G 中的边数。
- vertices(): 返回 G 中顶点的迭代器。
- edges(): 返回 G 中边的迭代器。

不像一棵树（它有根），图中没有特殊顶点。因此，有一个方法返回图中任意一个顶点，如下：

- aVertex(): 返回 G 的一个顶点。

以下访问方法以顶点位置和边位置作为参数：

- degree(v): 返回 v 的度。
- adjacentVertices(v): 返回与 v 相邻顶点的一个迭代器。
- incidentEdges(v): 返回依附 v 的边的一个迭代器。
- endVertices(e): 返回存储 e 的端点、且大小为2的数组。
- opposite(v, e): 返回 e 的不同于 v 的端点。
- areAdjacent(v, w): 返回顶点 v 和 w 是否相邻的判定。

2. 处理有向边的方法

如果图中某些边或者所有边有向，那么应该把下述方法包括在图ADT中。首先介绍与有向边有关的一些方法。

- directedEdges(): 返回所有有向边的一个迭代器。
- undirectedEdges(): 返回所有无向边的一个迭代器。

- `destination(e)`: 返回有向边 e 的目的点。
- `origin(e)`: 返回有向边 e 的源点。
- `isDirected(e)`: 当且仅当边 e 是有向的时, 返回真。

294

此外, 如果存在有向边, 需要根据方向把顶点和边关联的方式:

- `inDegree(v)`: 返回 v 的入度。
- `outDegree(v)`: 返回 v 的出度。
- `inIncidentEdges(v)`: 返回 v 的所有入边的一个迭代器。
- `outIncidentEdges(v)`: 返回 v 的所有出边的一个迭代器。
- `inAdjacentVertices(v)`: 沿着 v 的入边, 返回 v 的所有相邻顶点的一个迭代器。
- `outAdjacentVertices(v)`: 沿着 v 的出边, 返回 v 的所有相邻顶点的一个迭代器。

3. 图的更新方法

更新方法可以添加或删除边和顶点:

- `insertEdge(v, w, o)`: 在顶点 v 和 w 之间插入并返回一条无向边, 并把对象 o 存储在这个位置。
- `insertDirectedEdge(v, w, o)`: 在顶点 v 到顶点 w 之间插入并返回一条有向边, 并把对象 o 存储在这个位置。
- `insertVertex(o)`: 插入并返回一个新(孤立)顶点, 并把对象 o 存储在这个位置。
- `removeVertex(v)`: 删除顶点 v 以及所有依附它的边。
- `removeEdge(e)`: 删除边 e 。
- `makeUndirected(e)`: 建立一条无向边 e 。
- `reverseDirection(e)`: 把有向边 e 反向。
- `setDirectionFrom(e, v)`: 建立远离顶点 v 的有向边 e 。
- `setDirectionTo(e, v)`: 建立进入顶点 v 的有向边 e 。

诚然, 图ADT中还有许多方法, 但是, 不可避免地要对方法的数量进行某种程度扩展, 因为图具有如此丰富的结构。图支持两种位置——顶点和边, 甚至允许边是有向的或无向的。我们还需要访问和更新所有这些不同位置的方法, 以及处理存在于这些不同位置之间关系的方法。

295

6.2 图的数据结构

有几种利用具体数据结构实现图ADT的方式。在这一节里, 讨论常用方法。这些常用方法通常指的是边表(edge list)结构、邻接表(adjacency list)结构和邻接矩阵(adjacency matrix)结构。在这三种表示法中, 利用一个容器(例如, 一个表或向量)存储图中的顶点。关于边, 在前两种结构和后一种结构中存在实质性的差异。边表结构和邻接表结构只能存储图中实际上存在的边, 而邻接矩阵存储每一对顶点的占位符(不管其中是否有边相连)。正如在本节中所解释的那样, 这种差异蕴涵着, 对于具有 n 个顶点、 m 条边的图 G , 边表表示或邻接表表示所用空间为 $O(n+m)$, 而邻接矩阵表示所用空间为 $O(n^2)$ 。

6.2.1 边表结构

尽管不是最有效的, 但边表结构可能是图 G 的最简单的一种表示法。在这种表示法中, 存储元素 o 的 G 中的顶点 v 可用一个顶点对象显式表示。所有这样的顶点对象都存储在一个容器 v 中, 它可以是一个表、向量或者字典。例如, 如果把 V 表示为一个向量, 那么很自然地把每个顶点看作是编过号的。另一方面, 如果把 V 表示为字典, 那么, 很自然地把每个顶点标识为关联它的一个

关键字。注意，容器 V 中的元素是图 G 的顶点位置。

1. 顶点对象

对于存储元素 o 的顶点 v ，其顶点对象的实例变量有

- 指向 o 的引用
- 用于依附无向边数、输入有向边数及输出有向边数的计数器
- 指向容器 V 中顶点-对象的位置（或者定位器）的引用

边表结构的特殊性质不是它如何表示顶点，而在于它表示边的方式。在这个结构中，一个边对象显式地表示存储元素 o 的 G 中的边 e 。边对象存储在一个容器 E 中，它可以是一个表、向量或者字典（可能支持定位器模式）。

2. 边对象

对于存储元素 o 的边 v ，其边对象的实例变量有

- 指向 o 的引用
- 布尔指示器，表明 e 是有向边还是无向边 e
- 指向 V 中关联 e （如果边 e 是无向边）的端点的顶点对象或者关联 e （如果边 e 是有向边）的源点和目的点的顶点对象的引用
- 指向容器 E 中边对象的位置（或定位器）的引用

图6-3显示了有向图 G 的边表结构。

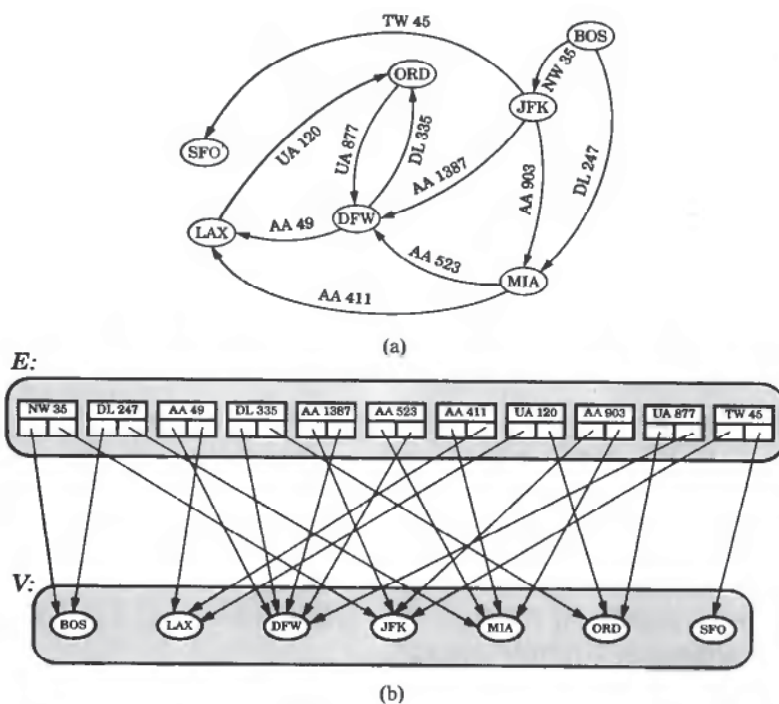


图6-3 (a)有向图 G ; (b) G 的边表结构示意图性表示。为避免混乱，并没有显示顶点对象的以下域：依附边的三个计数器、指向容器 V 中顶点-对象的位置（或定位器）的引用。同时也没有显示边对象的以下域：布尔方向指示器、指向容器 E 中边对象的位置（或定位器）的引用。最后，用元素名把存储在顶点对象和边对象中的元素可视化，而不是使用指向元素对象的实际引用

3. 边表

之所以称为边表 (edge list), 是因为实现容器 E 的最简单、最常用结构是用表完成的。为了能够方便地查找关联某些边的特定对象, 尽管称此为“边表”, 还是希望用字典实现 E 。出于同样的原因, 还希望用字典实现容器 V 。尽管如此, 为了与传统术语保持一致, 称这个结构为边表结构。

边表结构的主要特点是, 可以直接从边访问它们所依附的顶点。使得可以定义实现基于边的图ADT的各种方法 (例如, 方法 `endVertices`、`origin` 和 `destination`)。可以通过简单访问给定边对象的相应分量, 实现每个这样的方法。

然而, “逆”操作——访问依附顶点的边——要求穷尽查找 E 中的所有边。因此, `incidentEdges(v)` 的运行时间与图中的边数成正比, 与顶点 v 的度则不成比例, 如我们所想到的那样。事实上, 即使用 `areAdjacent(v, w)` 方法检查两个顶点 v 和 w 是否相邻时, 也要求在整个边表中查找边 (v, w) 或 (w, v) 。而且, 因为删除一个顶点涉及删除所有它的依附边, 所以方法 `removeVertex` 也要求对边表 E 进行完全查找。

4. 性能

假设 V 和 E 用双向链表实现, 边表结构包括如下重要的性能特点:

- 通过访问关联表 V 和/或表 E 的大小域, 可用 $O(1)$ 时间实现方法 `numVertex()`、`numEdges()` 和 `size()`。
- 与每个顶点对象存储在一起的计数器允许用常数时间执行方法 `degree`、`inDegree` 和 `outDegree`。
- 方法 `vertices()` 和 `edges()` 分别返回顶点表或边表的迭代器。同样, 可以扩展表 E 的迭代器, 使其仅返回那些具有正确类型的边, 实现迭代器 `directedEdges()` 和 `undirectedEdges()`。
- 因为容器 V 和 E 是用双向链表实现的表, 可用 $O(1)$ 时间实现插入顶点、插入边和删除边的操作。
- 方法 `incidentEdges`、`inIncidentEdge`、`outIncidentEdges`、`adjacentVertices`、`inAdjacentVertices`、`outAdjacentVertices` 和 `areAdjacent` 必须检查所有边, 才能确定哪条边依附哪个顶点, 所需时间均为 $O(m)$ 。
- 更新方法 `removeVertex(v)` 需要检查所有边, 才能找到并删除那些依附 v 的边, 所需时间为 $O(m)$ 。

298

6.2.2 邻接表结构

边表结构简单, 但有其局限性。因为许多方法本该可以快速查找单个顶点, 却必须要检查整个边表结构, 才能执行正确。图的邻接表 (adjacency list) 结构扩展了边表结构, 增加了支持对每个顶点依附边 (因而对邻接顶点) 进行直接访问的信息。边表结构仅从边的角度看待边-顶点的依附关系, 而邻接表结构从两个角度看待。这种对称性使得可以利用邻接表结构实现图ADT的许多顶点方法。即使这两种表示都利用了与图中的顶点数和边数成正比的空间量, 邻接表结构还是要比边表结构快得多。邻接表结构包括边表结构的所有结构成分, 此外, 还包括:

- 顶点对象 v 保存指向容器 $I(v)$ 的引用, 该容器称为依附容器 (incidence container), 它存储指向依附 v 的边的引用。如果边是有向的, 则将 $I(v)$ 分成 $I_{in}(v)$ 、 $I_{out}(v)$ 和 $I_{un}(v)$, 分别存储依附 v 的入边、出边和无向边。
- 在依附容器 $I(u)$ 和 $I(v)$ 中, 边 (u, v) 的边对象保存指向边的位置 (或定位器) 的引用。

邻接表

传统上, 用表实现一个顶点 v 的依附容器 $I(v)$, 这就是为什么称这种图的表示为邻接表(adjacency list)结构的原因。然而, 还有一些环境, 需要把依附容器 $I(v)$ 表示为(比如说)字典或优先队列。因此, 坚持把 $I(v)$ 看作边对象的泛型容器。如果支持图的一种表示既能表示包含有向边的图, 又能表示包含无向边的图, 那么对于每个顶点, 有三个依附容器 $I_{in}(v)$ 、 $I_{out}(v)$ 和 $I_{un}(v)$, 分别存储指向关联依附顶点 v 的有向入边、有向出边和无向边的边对象的引用。

邻接表结构可以直接从边访问顶点和从顶点访问它们的依附边。利用邻接表结构而不是边表结构, 由于能够在两个方向访问顶点和边, 这使得能够提高大量图方法的性能。图6-4说明了有向图的邻接表结构。对于顶点 v , v 的依附容器所用空间与 v 的度成正比, 即 $O(\deg(v))$ 。因此, 由定理6.1可知, 邻接表结构的空间需求为 $O(n + m)$ 。

299

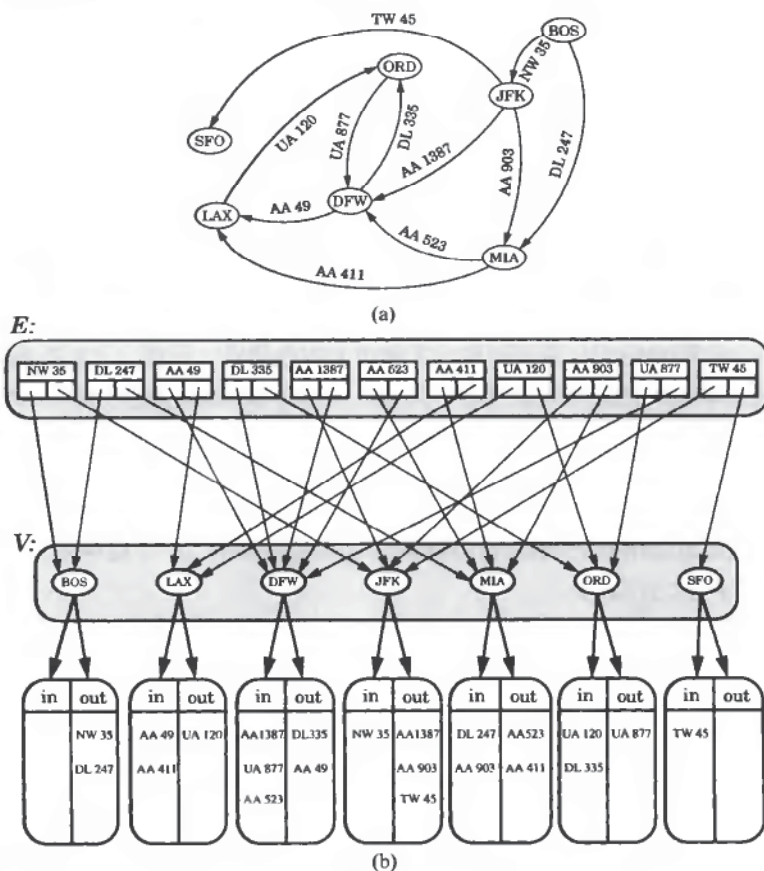


图6-4 (a)有向图 G ; (b) G 的邻接表结构的示意图表示。如在图6-3中所示的那样, 可用名字把容器中的元素可视化。同时, 只显示了有向边的依附容器, 因为这个图中没有无向边

邻接表结构与边表的性能匹配, 还提供如下方法的改进运行时间:

- 返回顶点 v 的依附边或邻接顶点迭代器的方法, 运行时间与其输出规模 $O(\deg(v))$ 成正比。
- 通过检查 u 或 v 的依附容器, 执行方法 $\text{areAdjacent}(u, v)$ 。选择这两者中的较小者, 可得 $O(\min\{\deg(u), \deg(v)\})$ 的运行时间。
- 方法 $\text{removeVertex}(v)$ 需调用 $\text{incidentEdges}(v)$, 确定要删除的边, 作为操作的结果。后续 $\deg(v)$ 条边中每条边的删除时间为 $O(1)$ 。

300

6.2.3 邻接矩阵结构

像邻接表结构一样，图的邻接矩阵表示还用另一个分量扩展了边结构。在这种情况下，用一个矩阵（二维数组） A 增大边表。这使可以在常数时间内，确定顶点对之间的邻接关系。正如将看到的那样，以数据结构的使用空间达到这种性能上的提高。

在邻接矩阵的表示中，将顶点编号为 $0, 1, \dots, n-1$ ，把边看作是这样的整数对。将图 G 表示成 $n \times n$ 数组 A ，满足如果边 (i, j) 存在，则 $A[i, j]$ 存储指向边的引用。如果边 (i, j) 不存在，则 $A[i, j]$ 为空。

确切地讲，邻接矩阵在如下几个方面扩展了边表结构：

- 顶点对象 v 还存储范围 $0, 1, \dots, n-1$ 内的不同整型关键字，称为 v 的下标（index）。为了简化讨论，称下标为 i 的顶点为“顶点 i ”。
- 保存二维 $n \times n$ 数组 A ，满足单元 $A[i, j]$ 存放指向依附顶点 i 和 j 的边的引用，如果这样的边存在。如果连接顶点 i 和 j 的边 e 存在，且是无向的，那么在 $A[i, j]$ 和 $A[j, i]$ 中同时存储指向 e 的引用。如果不存在从顶点 i 到顶点 j 的边，那么 $A[i, j]$ 引用空对象（或者指示该单元不与任何边关联的其他指示器）。

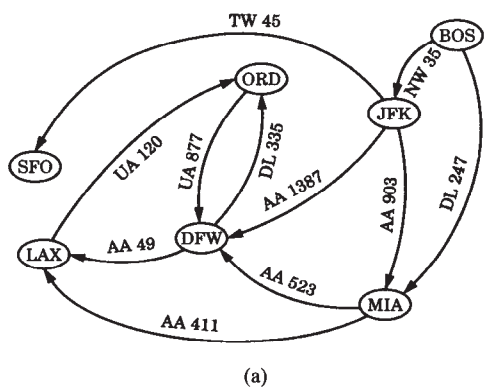
利用邻接矩阵 A ，执行方法 $\text{areAdjacent}(u, v)$ 的时间为 $O(1)$ 。通过访问顶点 v 和 w ，确定它们各自的下标 i 和 j ，达到这个性能。但是，达到这个性能是以增加空间（现在是 $O(n^2)$ ）和其他方法的运行时间为代价的。例如，诸如 incidentEdges 和 adjacentVertices 之类的方法需要检查数组 A 的整行和整列，这需要 $O(n)$ 时间。在空间使用上，邻接表结构优于邻接矩阵结构；除了方法 areAdjacent 之外，邻接表结构的所有其他方法具有更好的时间性能。

历史上，邻接矩阵是图的第一种表示方法，邻接矩阵被严格定义成布尔矩阵，如下：

$$A[i, j] = \begin{cases} 1 & \text{如果}(i, j)\text{是一条边} \\ 0 & \text{其他情况} \end{cases}$$

因此，邻接矩阵具有自然的数学结构（例如，无向图的邻接矩阵是对称的）。我们关于邻接矩阵的定义更新了关于面向对象框架的历史观点。当利用邻接矩阵表示一个图时，我们考察的大多数图算法都将最有效地运行。但是，在某些情况下，还要做出一些折中，这取决于图中的边数。

图6-5说明了邻接矩阵的一个例子。



(b)

0	1	2	3	4	5	6
BOS	DFW	JFK	LAX	MIA	ORD	SFO

(c)

	0	1	2	3	4	5	6
0	∅	∅	NW 35	∅	DL 247	∅	∅
1	∅	∅	∅	AA 49	∅	DL 335	∅
2	∅	AA 1387	∅	∅	AA 903	∅	TW 45
3	∅	∅	∅	∅	∅	UA 120	∅
4	∅	AA 523	∅	AA 411	∅	∅	∅
5	∅	UA 877	∅	∅	∅	∅	∅
6	∅	∅	∅	∅	∅	∅	∅

图6-5 邻接矩阵结构的示意图表示：(a)有向图 G ；(b)顶点编号；(c) G 的邻接矩阵 A

6.3 图的遍历

遍历 (traversal) 是通过检查图中的所有顶点和边来探索一个图的系统过程。例如, 一个Web蜘蛛 (spider) 或者爬虫 (crawler) 是搜索引擎中的数据收集部分, 通过考察图中表示文档的顶点以及表示文档之间超链接的边来探索超文本文档的图。如果一个遍历对所有顶点和边的访问与其数量成正比, 即具有线性时间, 则遍历是有效的。

6.3.1 深度优先查找

本章考虑的第一个遍历算法是深度优先查找 (depth-first search, DFS)。深度优先查找在关于图的许多计算中应用广泛。包括找出一条从某个顶点到另一个顶点的路径, 确定一个图是否是连通的, 以及计算一个连通图的最小生成树。

1. 用回溯技术遍历图

无向图 G 中的深度优先查找应用回溯 (backtracking) 技术, 这类似于在一个迷宫中拿着一根细绳和一桶油漆漫游而不会迷失方向。从 G 中的一个起始顶点 s 起始, 将细绳的一端固定到 s , 并将 s 着色为“访问过”。顶点 s 现在是“当前”顶点——称为当前顶点 u 。通过考虑依附当前顶点 u 的任意一条边 (u, v) 遍历 G 。如果边 (u, v) 通向一个已经访问过 (着色过) 的顶点 v , 那么, 直接回到顶点 u 。另一方面, 如果边 (u, v) 通向一个未被访问过的顶点 v , 则展开细绳, 并到达 v 。然后, 将 v 着色为“访问过”, 使它成为当前顶点, 重复上述计算。最终到达“终结顶点”, 即到达一个当前顶点 u , 满足依附 u 的所有边通向的顶点都已被访问过。因此, 取依附 u 的任何边都会使我们回到 u 。为了摆脱这种僵局, 沿着细绳返回, 并沿着把我们带到 u 的边回溯, 返回到前一个访问过的顶点 v 。然后使 v 成为当前顶点, 并对于任何依附 v 的、以前尚未考虑的边, 重复上述计算。如果 v 的所有依附边都通往访问过的顶点, 那么再次沿着细绳, 并回溯到从此到达 v 的顶点, 然后在那个顶点重复这个过程。因此, 沿着到目前为止经过的路径继续回溯, 直到找到一个具有尚未探索过的边的顶点, 在这一点上, 取这样一条边, 继续这个遍历。当所有回溯回到起始顶点 s 时, 不存在依附 s 的尚未探索过的边。这个简单遍历 G 中边的过程是非常精致和系统的。如图6-6所示。

303

2. 可视化深度优先查找

可以通过沿着遍历过程中探索各条边的方向对边进行定向, 来可视化DFS遍历的过程, 并把用于发现新顶点的边称为发现边 (discovery edge) 或树边 (tree edge); 而把那些通向访问过的顶点的边为后边 (back edge)。如图6-6所示。在上述的类比中, 发现边就是遍历时在其上展开细绳的那些边, 而后边则是那些不展开细绳而直接返回的边。发现边构成起始顶点 s 的连通分量的一棵生成树, 称为DFS树 (DFS tree)。称不在DFS树中的边为“后边”的原因是, 假设DFS树以起始顶点为根, 每条这样的边从此树中的一个顶点返回到它在树中的一个祖先顶点。

3. 递归深度优先查找

从顶点 v 开始的DFS遍历的伪代码类似于我们的细绳回溯技术。利用递归实现这个方法。假设有一种机制 (类似于喷漆) 用于确定一个顶点或边是否已被探索过, 并把边标记为发现边或是后边。这个DFS的伪代码描述如算法6-1所示。

对于深度优先查找算法, 有大量观察结果。许多结果源于DFS算法划分边的方式, 即DFS算法如何把无向图 G 划分成两组, 即发现边和后边。例如, 因为后边总是连接顶点 v 和前一个访问过的顶点 u , 每条后边蕴涵 G 中的一个回路, 它由从 u 到 v 的发现边以及后边 (u, v) 组成。

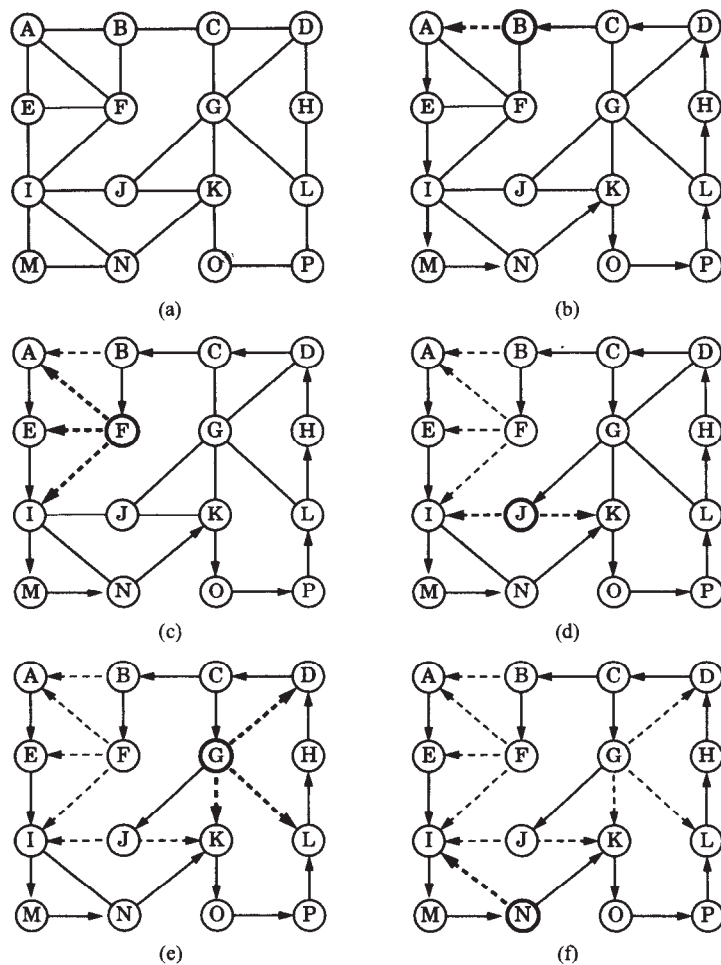


图6-6 从顶点A开始进行图的深度优先遍历的例子。实线表示发现边，虚线表示后边。加粗线表示当前顶点：(a)输入图；(b)从A回溯发现边的路径，直到遇见后边(B, A)；(c)到达F，它是一个终结顶点；(d)回溯到C后，假设边为(C, G)，遇见另一个终结顶点J；(e)回溯到G后；(f)回溯到N后

算法6-1 DFS算法的递归描述

算法 DFS(G, v):

输入: 图 G 和 G 中的顶点 v

输出: 把 v 的连通分量中的边标记为发现边和后边

标记 v 已访问过

for $G.\text{incidentEdges}(v)$ 中的所有边 e do

if 边 e 未被探索过 then

$w \leftarrow G.\text{opposite}(v, e)$

if 顶点 w 未被探索过 then

将边 e 标记为发现边

递归调用DFS(G, w)

else

将边 e 标为后边

定理6.5 设 G 是一个无向图，在其上已经执行以顶点 s 为起点的DFS遍历。那么遍历将访问 s 的连通分量中的所有顶点，并且发现边将构成 s 的连通分量的一棵生成树。

证明 用反证法证明。假定 s 的连通分量中至少存在一个顶点未被访问。设 w 是从 s 到 v （可能有 $v=w$ ）的某条路径上第一个未被访问的顶点。因为 w 是这条路径上第一个未被访问的顶点，所以它有一个近邻 u 已被访问。但是当访问 u 时，一定会考虑边 (u, v) ；因此，若 w 未被访问，则不可能是正确的。于是， s 的连通分量中不存在未被访问的顶点。因为在到达未被访问的顶点时，只会标记边，因而，永不会形成发现边构成的回路，即发现边永远不会构成一棵树。进而，这是一棵生成树，因为深度优先查找会访问 s 的连通分量中的每个顶点。 ■

注意：DFS在每个顶点上恰好会被调用一次，并且每条边恰好会被检查两次，即从它的两个端点各检查一次。设 m_s 表示顶点 s 的连通分量中的边数。假定满足以下条件，则开始于 s 的DFS的运行时间为 $O(m_s)$ ：

- 数据结构表示的图具有如下性能：
 - 方法`incidentEdges(v)`的时间为 $O(\deg(v))$ 。
 - `incidentEdges(v)`返回的`EdgeIterator`中的方法`hasNext()`和`nextEdge()`的运行时间均为 $O(1)$ 。
 - 方法`opposite(v, e)`的运行时间为 $O(1)$ 。
- 邻接表结构满足这些性质；而邻接矩阵则不满足。
- 有一种在 $O(1)$ 时间内标记顶点或边已探索过并且测试一个顶点或边是否已被探索过的方式。执行这种标记的一种方式是在扩展结点位置的功能，在顶点或边的实现中包含一个`visited`标志。另一种方式是利用修饰设计模式，这将在6.5节中讨论。

定理6.6 设 G 是有 n 个顶点、 m 条边的图，并用邻接表结构表示它。 G 的DFS一个遍历可用 $O(n+m)$ 时间完成。同时，对于以下问题，存在基于DFS的运行时间为 $O(n+m)$ 的算法：

- 测试 G 是否连通。
- 计算 G 的生成森林。
- 计算 G 的连通分量。
- 计算 G 的两个顶点之间的一条路径，或报告 G 中不存在这样的路径。
- 计算 G 中的一个回路，或报告 G 中不存在回路。

306

在几个习题中探讨了这一定理的证明细节。

6.3.2 双连通分量

设 G 是一个连通图， G 的一条孤立边（separation edge）是一条删除它会导致 G 不连通的边。孤立顶点（separation vertex）是一个删除它会导致 G 不连通的顶点。孤立边和孤立顶点对应于网络中的单点故障；因此，常常希望将它们找出来。如果对于 G 中的任意两个顶点 u 和 v ，在 u 和 v 之间存在两条不相交的路径，即除 u 和 v 之外，两条路径没有共享公共边或顶点，则称连通图 G 是双连通的（biconnected）。 G 的双连通分量（biconnected component）是一个满足以下条件之一的子图（如图6-7所示）：

- G 的子图是双连通的，并且对于这个子图，增加任何额外的顶点或边都会强制它成为非双连通的
- 由孤立边及其端点组成 G 中的一条边

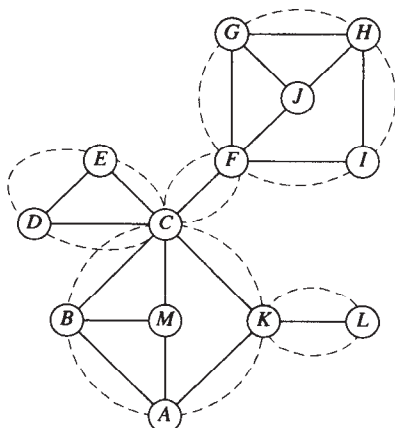


图6-7 虚线包含的双连通分量， C 、 F 和 K 是孤立顶点； (C, F) 和 (K, L) 是孤立边

如果 G 是双连通的，它有一个双连通分量，即 G 自身。另一方面，如果 G 中没有回路，那么 G 中的每一条边都是一个双连通分量。在计算机网络中，双连通分量是一个重要的概念，其中顶点表示路由器，边表示连接关系，因为即使连通分量中的一个路由器发生故障，消息仍然可用其他路由器在那个分量中路由。

如以下引理（其证明留作习题（C-6.5））所指出的，双连通性等价于不存在孤立顶点和孤立边。

引理6.1 设 G 是一个连通图。以下陈述是等价的：

- (1) G 是连通的。
- (2) 对于 G 中任意两个顶点，存在包含它们的一个简单回路。
- (3) G 中不存在孤立顶点或孤立边。

307

1. 等价类和连接关系

给定对象集 C ，对于 C 中的每一对 x 和 y ，定义布尔关系 $R(x, y)$ ，即对于 C 中的每一对 x 和 y ，将 $R(x, y)$ 定义为真或假。如果关系 R 具有以下性质，则关系 R 是等价关系（equivalence relation）。

- 自反性（reflexive property）：对于 C 中的每一个 x ， $R(x, x)$ 为真。
- 对称性（symmetric property）：对于 C 中的每一对 x 和 y ， $R(x, y) = R(y, x)$ 。
- 传递性（transitive property）：对于 C 中的每个 x 、 y 和 z ，如果 $R(x, y)$ 为真且 $R(y, z)$ 为真，那么 $R(x, z)$ 也为真。

例如，通常的“相等”运算符（ $=$ ）是任何数集合上的等价关系。 C 中任何对象 x 的等价类（equivalence class）是所有满足 $R(x, y)$ 为真的对象 y 的集合。注意，集合 C 的任一等价关系 R 把集合 C 划分成由 C 中对象的等价类组成的不相交子集。

可以定义图 G 的各条边上的有趣的连接关系（link relation）。如果 G 中的两条边 e 和 f 满足 $e = f$ 或 G 具有包含 e 和 f 的一个简单回路，则称它们是连接的（linked）。以下引理给出连接关系的基本性质。

引理6.2 设 G 是一个连通图。那么，

- (1) 连接关系构成 G 中各条边上的一个等价关系。
- (2) G 的一个双连通分量是由连接边的等价类导出的子图。
- (3) 当且仅当 e 形成连接边的单元元素等价类时， G 的一条边 e 是一条孤立边。
- (4) 当且仅当 v 在至少两个不同连接边的等价类中有依附边时， G 的顶点 v 是一个孤立顶点。

证明 容易看出, 连接关系具有自反性和对称性。为了证明它的传递性, 假定边 f 和 g 是连接的, 边 g 和 h 是连接的。如果 $f=g$ 或者 $g=h$, 那么 $f=h$, 或者存在包含 f 和 h 的简单回路; 因此, f 和 h 是连接的。假定 f 、 g 和 h 互不相同。也就是, 存在一条通过 f 和 g 的简单回路 C_{fg} , 以及一条通过 g 和 h 的简单回路 C_{gh} 。考虑回路 C_{fg} 和回路 C_{gh} 的并集所得的图。虽然这个图自身可能不是一个简单的回路 (尽管可能有 $C_{fg}=C_{gh}$), 但是它包括通过 f 和 h 的一条简单的回路 C_{fh} 。因此, f 和 h 是连接的。于是, 连接关系是一个等价关系。

308 连接关系等价类和 G 的双连通分量之间的对应关系可由引理6.1得到。 ■

2. 用DFS计算双连通分量的连接方法

因为 G 中边的连接关系等价类与双连通分量相同, 由引理6.2可知, 要构造 G 的双连通分量, 只需要计算 G 的边之间的连接关系的等价类。为了进行这个计算, 首先用DFS遍历 G , 并构造辅助图 (auxiliary graph) B , 如下:

- B 中的顶点是 G 的边。
- 对于 G 中的每条后边 e , 设 f_1, \dots, f_k 是 G 中形成含 e 的一个回路的发现边。图 B 包含边 $(e, f_1), \dots, (e, f_k)$ 。

309 因为存在 $m-n+1$ 条后边, 并且后边导出的每个回路至多有 $O(n)$ 条边, 因而, 图 B 至多有 $O(nm)$ 条边。

3. $O(nm)$ 时间的算法

从图6-8可知, 似乎 B 中的每个连通分量对应于图 G 的连接关系的一个等价类。毕竟, 对于在包含 f 的回路中找到的后边 e , 由 e 和DFS生成树导出一条边 (e, f) , 并将其添加在 B 中。

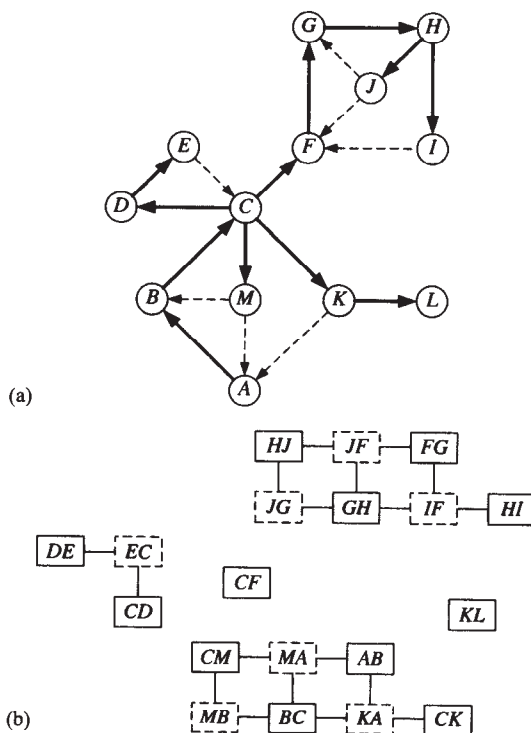


图6-8 用于计算连接分量的辅助图: (a)在图 G 上进行了DFS遍历 (虚线表示后边); (b)关联 G 及其DFS遍历的辅助图

以下引理（其证明留作习题（C-6.7））建立了图 B 和 G 中关于 G 的分量上连接关系的等价类之间的一个稳固的关系。为简明起见，称连接关系中的这个等价类为 G 的连接分量（link component）。

引理6.3 辅助图 B 的连通分量对应于导出 B 的图 G 的连接分量。

引理6.3产生以下计算具有 n 个顶点、 m 条边的图 G 的所有连接分量的算法，该算法运行时间为 $O(nm)$ 。

- (1) 对 G 进行DFS遍历 T 。
- (2) 计算辅助图 B ，找出关于 T 的每条后边所导出的 G 的回路。
- (3) 计算 B 的连通分量，例如，通过对辅助图 B 进行DFS遍历。
- (4) 对于 B 的每个连通分量，输出 B 中的顶点（它们是 G 中的边），作为 G 的连接分量。

通过确定 G 中的连接分量，可以用线性时间确定图 G 的双连通分量、孤立顶点和孤立边，即把 G 中的边划分成关于连接关系的等价类后，利用引理6.2中列出的简单规则，可用 $O(n+m)$ 时间找出 G 的双连通分量、孤立顶点和孤立边。不幸的是，构造辅助图 B 的时间多达 $O(nm)$ ；因此，这个算法中的瓶颈计算是 B 的构造。

但是注意，要找出 G 的双连通分量实际上并不需要 B 的所有辅助图。只需确定 B 中的连通分量。因此，只要简单地计算 B 中每个连通分量的一个生成树就足够了，即计算 B 的生成森林即可。因为 B 的生成森林中的连通分量与图 B 自身相同，实际上不需要 B 中的所有边——只需要它们足以构造 B 的一个生成森林即可。

于是，我们集中关注如何把这个更有效的生成森林方法应用于计算 G 中关于连接关系的边的等价类。

310

4. 线性时间的算法

如上所述，通过利用更小规模的辅助图，可以把计算图 G 的连接分量所需的时间减少到 $O(m)$ 。这个辅助图是 B 的一个生成森林。算法6-2描述了这个算法。

算法6-2 用于计算连接分量的线性时间的算法。注意 F 中由单个“未连接”顶点组成的连通分量对应于一条孤立边（在连接关系中，只与自己相关）

算法 LinkComponents(G):

输入：连通图 G

输出： G 的连接分量

设 F 是初始为空的辅助图。

从任一顶点 s 开始，执行对于 G 的DFS遍历。

添加每条DFS发现边 f 作为 F 中的一个顶点，标记 f “未连接”。

对于 G 中的每个顶点 v ，设 $p(v)$ 是DFS生成树中 v 的父顶点。

for 每个顶点 v ，在DFS遍历中，按照递增位序访问它们**do**

for 目的点为 v 的每一条后边 $e = (u, v)$ **do**

 把 e 添加到图 F 中，作为一个顶点。

 {从 u 行进到 s ，只在需要时，向 F 中添加边。}

while $u \neq v$ **do**

 设 f 是 F 中的顶点，对应于发现边 $(u, p(u))$ 。

 向 F 中添加边 (e, f) 。

if f 标记为“未连接” **then**

 将标记 f 为“连接”。

$u \leftarrow p(u)$

else

$u \leftarrow v$ {到达while循环末尾的捷径}
计算图 F 的连通分量

由算法6-2分析 LinkComponents 的运行时间。 G 的初始 DFS 遍历需要 $O(m)$ 时间。但是，主要计算是构造辅助图 F ，它所需的时间与 F 中的顶点数和边数成正比。注意：在算法执行的某个点处，把 G 的每一条边添加作为 F 的一个顶点。利用会计支付方法处理 F 中的边，即每次从一条新遇见的后边 e 到一条发现边 f ，向 F 中添加一条边 (e, f) ，如果 f 标记为“未连接”，则为此操作支付 f ，否则支付 e 。由内层 while 循环的构造可见，在利用这种模式的算法执行过程中，至多为 F 的每个顶点支付一次。因而，构造 F 所需的时间为 $O(m)$ 。最终， F 的连通分量的计算对应 G 的连接分量的计算，所需时间为 $O(m)$ 。

在 LinkComponents 中，图 F 是引理6.3中提到的图 B 的一个生成森林，由这一事实可知上述算法是正确的。有关细节，见习题C-6.8。于是，可以总结为以下定理，图6-9给出了 LinkComponents 的一个例子。

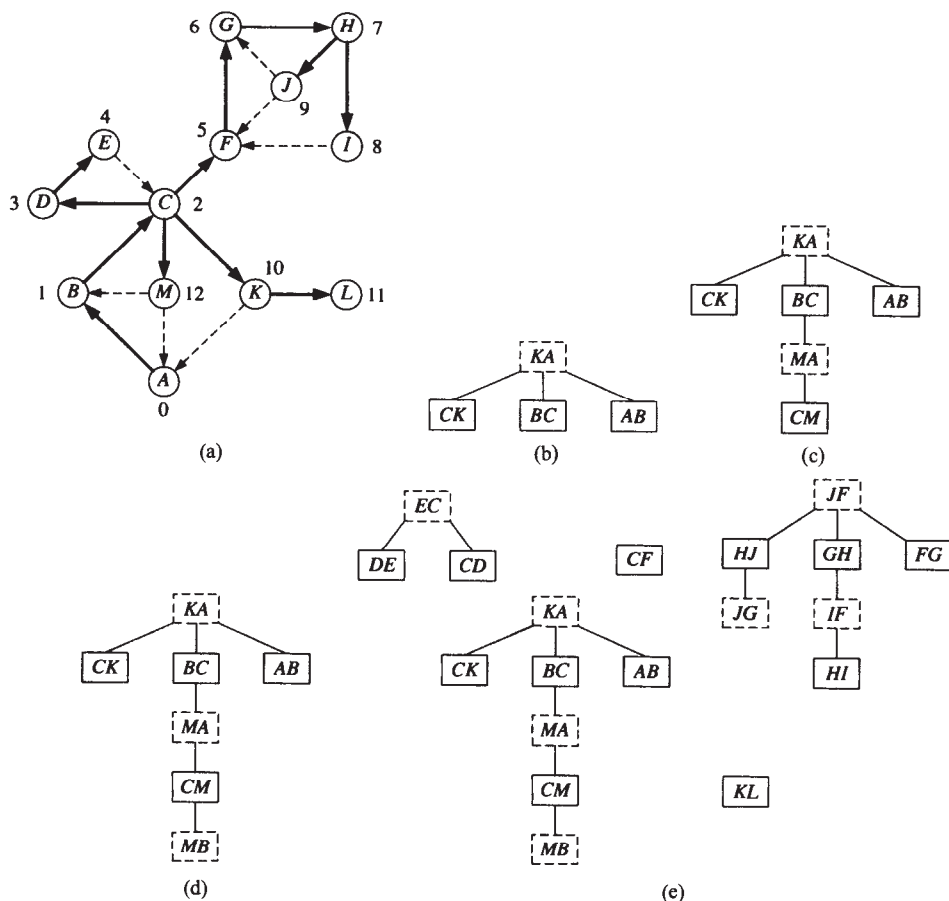


图6-9 算法LinkComponents (算法6-2) 执行示例: (a)DFS遍历之后的输入图 G (按照访问的次序用其位序标记顶点, 虚线表示后边); 处理后的辅助图 F ; (b)后边 (K, A) ; (c)后边 (M, A) ; (d)后边 (M, B) ; (e)算法结束时的图 F

定理6.7 给定一个具有 m 条边的连通图 G ，计算 G 的双连通分量、孤立顶点和孤立边的时间为 $O(m)$ 。

6.3.3 广度优先查找

在这一节里，考虑广度优先查找（breadth-first search, BFS）算法。像DFS算法一样，BFS遍历图的一个连通分量，且在之一过程中，定义了一棵有用的生成树。然而，BFS不像DFS那样“冒险”。BFS不是在图中漫游，而是按照轮数并把顶点分成层（level）。可把BFS看作用一根细绳和颜料进行遍历，BFS用一种更“保守”的方式展开细绳。

BFS从某个顶点 s 开始，该顶点位于第0层，并为我们的细绳定义了一个“锚”。在第一轮中，展开细绳作为一条边的长度，并访问所有能够达到的顶点，而无需进一步展开细绳。在这种情况下，访问与起始顶点 s 相邻并标记为“访问过”的顶点，这些顶点放在第1层上。在第二轮中，展开细绳作为两条边的长度，并访问所有能够达到的顶点，而无需进一步展开细绳。这些与第1层中的顶点相邻的新顶点以前不属于任何一层，现在则被放在第2层中，依此类推。当每个顶点都被访问过时，BFS遍历终止。

从顶点 s 开始的BFS遍历的伪代码如算法6-3所示。利用辅助空间标记边，标记访问过的顶点，并存储关联层的容器，即容器 L_0 、 L_1 、 L_2 等分别存储第0层、第1层、第2层等中的结点。例如，这些容器可实现为队列。它们也允许BFS成为一个非递归的算法。

算法6-3 图的BFS遍历

```

算法 BFS( $G, s$ ):
    输入: 图 $G$ 和 $G$ 中的顶点 $s$ 
    输出: 把 $s$ 的连通分量中的边标记为发现边和交叉边
    建立空容器 $L_0$ 
    把 $s$ 插入容器 $L_0$ 中
     $i \leftarrow 0$ 
    while  $L_i$ 非空 do
        建立空容器 $L_{i+1}$ 
        for  $L_i$ 中的每个顶点 $v$  do
            for  $G.\text{incidentEdges}(v)$ 中的所有边 $e$  do
                if 边 $e$ 未被探索过 then
                    设 $w$ 是 $e$ 的另一个端点
                    if 顶点 $w$ 未被探索过 then
                        将边 $e$ 标记为发现边
                        把 $w$ 插入 $L_{i+1}$ 中
                else
                    将 $e$ 标记为交叉边
             $i \leftarrow i+1$ 
  
```

313

图6-10说明了BFS遍历的过程。

BFS方法的优美性质之一是，在进行BFS遍历的过程中，从顶点 s 开始，可以用最短路径的长度（根据边数得出）标记每个顶点。尤其是，如果从顶点 s 开始，BFS把顶点 v 放在第 i 层，那么从 s 到 v 的最短路径长度为 i 。

像DFS一样，可把BFS遍历的过程可视化。沿着遍历中探索边的方向，对边进行定向，并称那些用于找出新顶点的边为发现边（discovery edge），称那些通向已访问过顶点的边为交叉边（cross edge）（如图6-10f所示）。与DFS一样，发现边形成一棵生成树，在这种情况下称为BFS树（BFS tree）。但是，在这种情况下，并不称这些非树边为“后边”，因为这些边不会连接顶点与它

的一个祖先顶点。每个非树边都会连接顶点 v 与另一个顶点，该顶点既不是 v 的祖先顶点，也不是 v 的后代顶点。

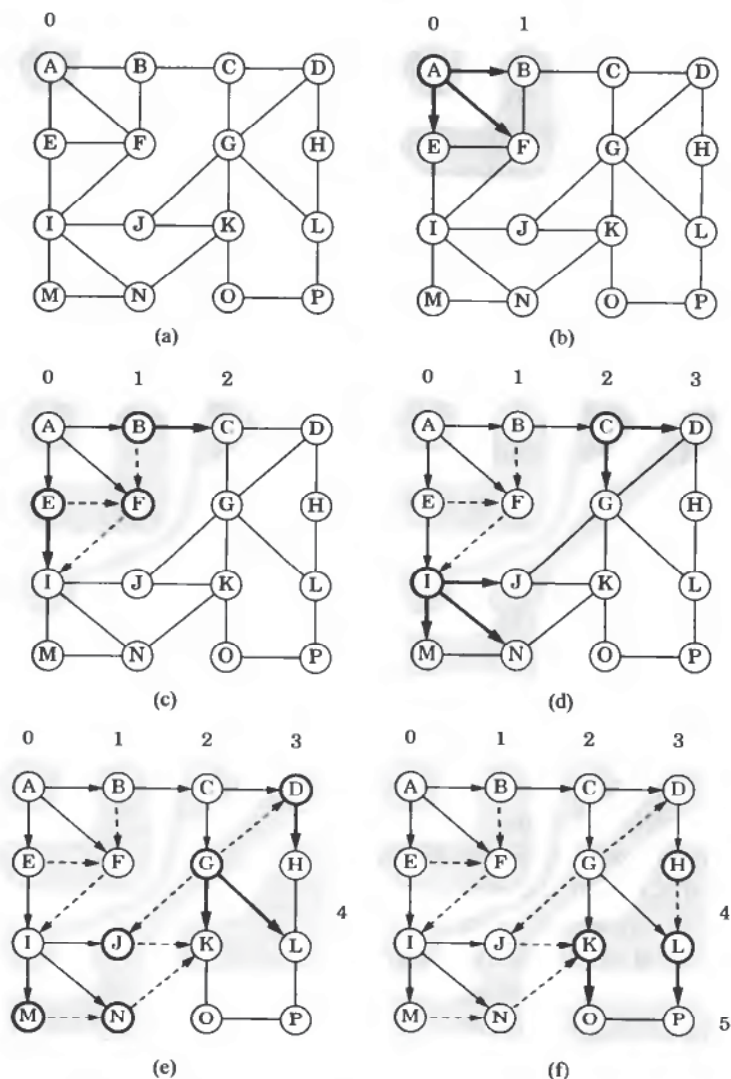


图6-10 广度优先遍历示例，按照邻接顶点的字母次序探索依附某个顶点的边。发现边用实线表示，交叉边用虚线表示：(a)遍历之前的图；(b)第1层发现；(c)第2层发现；(d)第3层发现；(e)第4层发现；(f)第5层发现

BFS遍历算法有大量有趣的性质，以下定理阐述了其中的一些性质。

定理6.8 设 G 是无向图，在其上从顶点 s 开始进行BFS遍历。那么：

- 遍历访问 s 的连通分量中的所有顶点。
- 发现边形成 s 的连通分量的一棵生成树 T 。
- 对于第 i 层上的每个顶点 v ，树 T 中顶点 s 和 v 之间的路径有 i 条边，图 G 中顶点 s 和 v 之间的任何其他路径至少有 i 条边。
- 如果 (u, v) 是一条交叉边，那么 u 和 v 之间的层数至多相差1。

这个定理的证明留作习题 (C-6.20)。BFS算法的运行时间的分析类似于DFS算法。

定理6.9 设 G 是具有 n 个顶点、 m 条边的图，用邻接表结构表示。 G 的BFS遍历所需时间为 $O(n+m)$ 。同时，存在基于BFS的运行时间为 $O(n+m)$ 的算法，用于求解下列问题：

- 测试 G 是否是连通的
- 计算 G 的一个生成森林
- 计算 G 的连通分量
- 给定 G 的起始顶点 s ，对于 G 中的每个顶点 v ，计算 G 中两个顶点 s 和 v 之间具有最少边数的路径，或者报告不存在这样的路径。
- 计算 G 中的一个回路，或者报告 G 中不存在回路。

314
315

比较BFS和DFS

由定理6.9可知，BFS遍历可以做DFS遍历所做的所有事情。但是，在这两种方法之间，存在大量有趣的差异，事实上，对于许多任务，一个遍历会比另一个遍历做得更好。在查找图的最短路径方面（其中距离按照边数度量），BFS遍历要做得更好一些。同时它还会产生一棵生成树，满足所有非树边是交叉边。在解答复杂的连通性问题方面，DFS遍历更好一些。例如，确定图中的每一对顶点是否可用两条不相交的路径连接。此外，产生一棵生成树，满足所有非树边都是后边。这些性质仅对于无向图成立。然而，在下一节里将会探讨，对于有向图，DFS和BFS也有大量类似的有趣性质。

6.4 有向图

在这一节里，考虑有向图中的一些特定问题。回忆，有向图也称为digraph，是一种其边都有方向的图。

可达性

有向图中的基本问题是可达性 (reachability) 的概念，它涉及确定可以在有向图中到达何处。例如，在一个单向连接的计算机网络中，比如涉及卫星连接，一个重要的问题是：给定网络中的一个结点，确定是否可从该结点到达网络中的其他结点。有向图中的遍历总是沿着有向路径，即沿着各自方向遍历所有边的路径。给定有向图 \vec{G} 中的顶点 u 和 v ，如果 \vec{G} 中存在从 u 到 v 的有向路径，则称 u 到达 v （或 v 从 u 是可达的）。如果 v 到达边的源点 w ，则称顶点 v 到达边 (w, z) 。

如果对于 \vec{G} 中的任意两个顶点 u 和 v ，满足 u 到达 v 并且 v 到达 u ，则称有向图 \vec{G} 是强连通的 (strongly connected)。 \vec{G} 中的一个有向回路 (directed cycle) 是一个回路，满足所有边按照它们各自的方向遍历（注意， \vec{G} 可能包含两条边组成的回路，这两条边方向相反，顶点相同）。如果有向图 \vec{G} 中不含有向回路，则称它是无环的 (acyclic)（如图6-11中的一些示例所示）。

有向图 \vec{G} 的传递闭包 (transitive closure) 是图 \vec{G}^* ，满足 \vec{G}^* 中的顶点与 \vec{G} 中的顶点相同，且 \vec{G}^* 中有一条边 (u, v) ，只要 \vec{G} 中存在一条从 u 到 v 的有向路径，即定义 \vec{G}^* 从有向图 \vec{G} 开始，对于每个 u 和 v ，向 \vec{G}^* 中添加一条额外的边 (u, v) ，满足在 \vec{G} 中 v 由 u 可达（ \vec{G} 中没有这条边 (u, v) ）。

316

有向图 \vec{G}^* 中涉及可达性的有趣问题如下：

- 给定顶点 u 和 v ，确定 u 是否达到 v 。
- 找出由给定顶点 s 可达的 \vec{G} 中的所有顶点。
- 确定 \vec{G} 是否是强连通的。
- 确定 \vec{G} 是否是无环的。

- 计算 \vec{G} 的传递闭包 \vec{G}^* 。

在本节余下内容中, 探讨求解这些问题的一些有效算法。

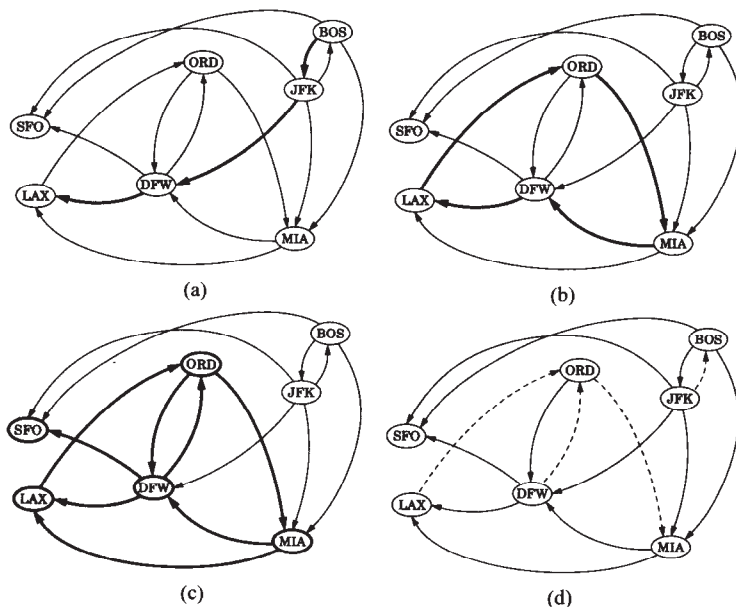


图6-11 有向图中的可达性示例: (a)用粗线表示从BOS到LAX的一条有向路径; (b)用粗线表示有向回路(ORD, MIA, DFW, LAX, ORD); 它的顶点导出一个强连通子图; (c)用粗线表示由ORD可达的顶点和边的子图; (d)删除虚线边得到一个无环有向图

317

6.4.1 遍历有向图

像无向图一样, 可以利用深度度优先查找 (DFS) 和广度优先查找 (BFS) 算法, 系统地查找一个有向图。这些查找算法在6.3.1节和6.3.3节已经针对无向图中作过定义。例如, 这样的探索方法可用于解答可达性问题。本节研究的有向图的深度优先查找方法和广度优先查找方法与无向图中相应的查找方法非常相似。事实上, 唯一真正的差别在于, 有向图的深度优先查找和广度优先查找方法仅按照它们各自的方向遍历边。

图6-12说明了一个开始于顶点 v 的有向图的DFS遍历过程。

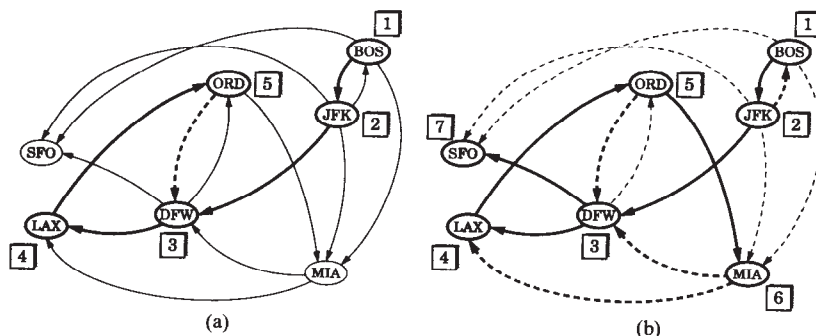


图6-12 有向图中的DFS示例: (a)中间步骤, 其中第一次到达一个已经访问过的顶点 (DFW); (b)完整的DFS。用粗实线表示发现边, 粗虚线表示后边, 细虚线表示前向边和交叉边。顶点被访问的次序用紧邻每个顶点的标号表示。边(ORD, DFW)是一条后边, 但是(DFW, ORD)是一条前向边。边(BOS, SFO)是一条前向边

在有向图 \vec{G} 上进行有向DFS, 把 \vec{G} 中从起始顶点可达的边划分为发现边或树边。这使得可以找到一个新顶点和非树边 (nontree edge), 通向一个以前访问过的顶点。发现边形成以起始顶点为根的树, 称为有向DFS树 (directed DFS tree)。同时, 把这三种类型的非树边分为 (如图6-12b所示):

- 后边, 把一个顶点和DFS树中它的一个祖先顶点连接起来。
- 前向边, 把一个顶点和DFS树中它的一个后代顶点连接起来。
- 交叉边, 把一个顶点和一个既不是它的祖先顶点也不是它的后代顶点的顶点连接起来。

318

定理6.10 设 \vec{G} 是一个有向图。 \vec{G} 上的从顶点 s 开始的深度优先查找会访问 \vec{G} 中所有由 s 可达的顶点。同时, DFS树包含从 s 到其每个可达顶点的有向路径。

证明 设 V_s 是DFS从顶点 s 开始遍历的 \vec{G} 中顶点的子集。希望证明 V_s 包含 s 及属于 V_s 的由 s 可达的每个顶点。用反证法证明。假定存在由 s 可达的顶点 w 不在 V_s 中。考虑由 s 到 w 的有向路径, 设 (u, v) 是这条路径上的第一条边, 满足 u 在 V_s 中, 而 v 不在 V_s 中。当DFS到达顶点 u 时, 探索 u 的所有出边, 从而必定经过边 (u, v) 到达顶点 v 。因此, v 应该在 V_s 中, 这与假设矛盾。因此, V_s 必定包含由 s 可达的每个顶点。 ■

有向DFS算法的运行时间分析类似于无向DFS算法的运行时间分析。每个顶点处只进行一次递归调用。每条边只会被遍历一次 (沿着它的方向)。因此, 如果由顶点 s 可达的子图有 m 条边, 那么从顶点 s 开始的有向DFS的运行时间为 $O(n_s + m_s)$, 假定用邻接表结构表示有向图。

由定理6.10可知, 可以利用DFS找出由给定顶点可达的所有顶点, 进而找出 \vec{G} 的传递闭包, 即可从 \vec{G} 的每个顶点 v 开始, 执行DFS过程, 可知哪个顶点 w 由 v 可达, 把边 (v, w) 添加到每个这样的顶点 w 的传递闭包中。同样, 依次从每个顶点开始, 用DFS反复遍历有向图 \vec{G} , 可以容易地测试出 \vec{G} 是否是强连通的。于是, 如果每个DFS都会访问 \vec{G} 中的所有顶点, 则 \vec{G} 是强连通的。

定理6.11 设 \vec{G} 是具有 n 个顶点、 m 条边的有向图。以下问题可用运行时间为 $O(n(n + m))$ 的算法求解:

- 对于 \vec{G} 中的每个顶点 v , 计算由 v 可达的子图
- 测试 \vec{G} 是否是强连通的
- 计算 \vec{G} 的传递闭包 \vec{G}^*

1. 强连通性测试

实际上, 可以用比 $O(n(n + m))$ 时间快得多的算法确定一个有向图 \vec{G} 是否是强连通的, 它们就是利用两种深度优先查找。

从任一顶点 s 开始, 对有向图 \vec{G} 执行DFS过程。如果 \vec{G} 中存在任一顶点未被这个DFS访问过, 并且这个顶点由 s 不可达, 那么图不是强连通的。因此, 如果第一次DFS访问了 \vec{G} 中的每个顶点, 那么, 把 \vec{G} 中的所有边反向 (利用reverseDirection方法), 在这个“反向”图上, 执行从 s 开始的另一次DFS过程。如果 \vec{G} 中顶点都被这个第二次的DFS过程访问过, 那么这个图是强连通的, 因为这个DFS访问的每个顶点都可以到达 s 。因为这个算法只进行两次对 \vec{G} 的DFS遍历, 所以它的运行时间为 $O(n + m)$ 。

319

2. 有向广度优先查找

像DFS那样, 可以扩展广度优先查找 (BFS), 使之适合于有向图。算法仍然逐层访问顶点, 并把边的集合划分成为树边 (或发现边) 和非树边, 这些树边构成一棵以起始顶点为根的有向广

度优先查找树。与有向DFS方法不同的是，有向BFS方法只会留下两种非树边：

- 后边，把一个顶点和它的一个祖先顶点连接起来。
 - 交叉边，把一个顶点和另一个既不是它的祖先顶点也不是它的后代顶点的顶点连接起来。
- 其中没有前向边，在习题（C-6.14）中探讨这一事实。

6.4.2 传递闭包

在这一节里，探讨计算有向图的传递闭包的另一种技术。也就是说，描述一个直接的方法，确定一个有向图中的所有顶点对 (v, w) ，满足 w 由 v 可达。例如，这样的信息在计算机网络中是有用的，因为从中可以快速得知，一条消息是否可从一个结点 v 路由到另一结点 w ，或者是否能说这条消息“不能从这儿到那儿”。

设 \vec{G} 是具有 n 个顶点、 m 条边的有向图。重复若干轮，计算出 \vec{G} 的传递闭包。初始化 $\vec{G}_0 = \vec{G}$ 。也可以将 \vec{G} 中的顶点任意编号为

$$v_1, v_2, \dots, v_n$$

从第一轮开始计算，反复进行。在常规的第 k 轮中，从 $\vec{G}_k = \vec{G}_{k-1}$ 开始构造有向图 \vec{G}_k ，如果有向图 \vec{G}_{k-1} 中包含边 (v_i, v_k) 和边 (v_k, v_j) ，则向 \vec{G}_k 中添加有向边 (v_i, v_j) 。用这种方式，得到一个简单的规则，由以下引理给出。

引理6.4 对于 $i = 1, \dots, n$ ，当且仅当有向图 \vec{G}_k 包含从 v_i 到 v_j 的有向路径，并且它的中间顶点（如果有）在集合 $\{v_1, \dots, v_k\}$ 中时，有向图 \vec{G}_k 包含边 (v_i, v_j) 。特别地， \vec{G}_n 等于 \vec{G} 的传递闭包 \vec{G}^* 。

这个引理给出了计算 \vec{G} 的传递闭包的一种简单动态规划（dynamic programming）算法（5.3节），称之为Floyd-Warshall算法。算法6-4给出了这个方法的伪代码。

容易分析Floyd-Warshall算法的运行时间。主循环执行 n 次，内循环则考虑 $O(n^2)$ 对顶点，并且每对顶点计算时间为常数。如果利用邻接矩阵结构作为数据结构，则支持方法areAdjacent和insertDirectedEdge的运行时间为 $O(1)$ ，因此，算法的总运行时间为 $O(n^3)$ 。

算法6-4 Floyd-Warshall算法。这个动态规划算法通过渐次计算一系列有向图 $\vec{G}_0, \vec{G}_1, \dots, \vec{G}_n$ ，计算 \vec{G} 的传递闭包 \vec{G}^* ，其中 $k = 1, \dots, n$

算法 FloydWarshall(\vec{G}):

输入：具有 n 个顶点的有向图

输出： \vec{G} 的传递闭包 \vec{G}^*

设 v_1, v_2, \dots, v_n 是 \vec{G} 中顶点的任意编号

$\vec{G}_0 \leftarrow \vec{G}$

for $k \leftarrow 1$ **to** n **do**

$\vec{G}_k \leftarrow \vec{G}_{k-1}$

for $i \leftarrow 1$ **to** n , $i \neq k$ **do**

for $j \leftarrow 1$ **to** n , $j \neq i, k$ **do**

if 边 (v_i, v_k) 和 (v_k, v_j) 都在 \vec{G}_{k-1} 中 **then**

if \vec{G}_k 中不含有向边 (v_i, v_j) **then**

 向 \vec{G}_k 中添加有向边 (v_i, v_j)

return \vec{G}_n

上述描述和分析蕴涵以下定理。

定理6.12 设 \vec{G} 是邻接矩阵结构表示的具有 n 个顶点的有向图。Floyd-Warshall算法计算 \vec{G} 的传递闭包 \vec{G}^* 的运行时间为 $O(n^3)$ 。

Floyd-Warshall算法的性能

现在把Floyd-Warshall算法的运行时间与定理6.11中更复杂的算法的运行时间作一比较。定理6.11中的算法从每个顶点开始，反复执行DFS算法 n 次。

如果用邻接矩阵结构表示有向图，那么一次DFS遍历所需的时间为 $O(n^2)$ （在习题中探讨其原因）。因此，执行 n 次DFS算法的运行时间为 $O(n^3)$ ，这并不比单独执行一次Floyd-Warshall算法好。

如果用邻接表结构表示有向图，那么，执行 n 次DFS算法的运行时间为 $O(n(n+m))$ 。即使如此，如果图是稠密的（dense），即如果有 $\Theta(n^2)$ 条边，那么这种方法的运行时间仍然为 $O(n^3)$ 。

因此，仅当图不是稠密的且利用邻接表结构表示时，定理6.11中的算法才好于Floyd-Warshall算法。

图6-13说明了Floyd-Warshall算法的一个运行示例。

321

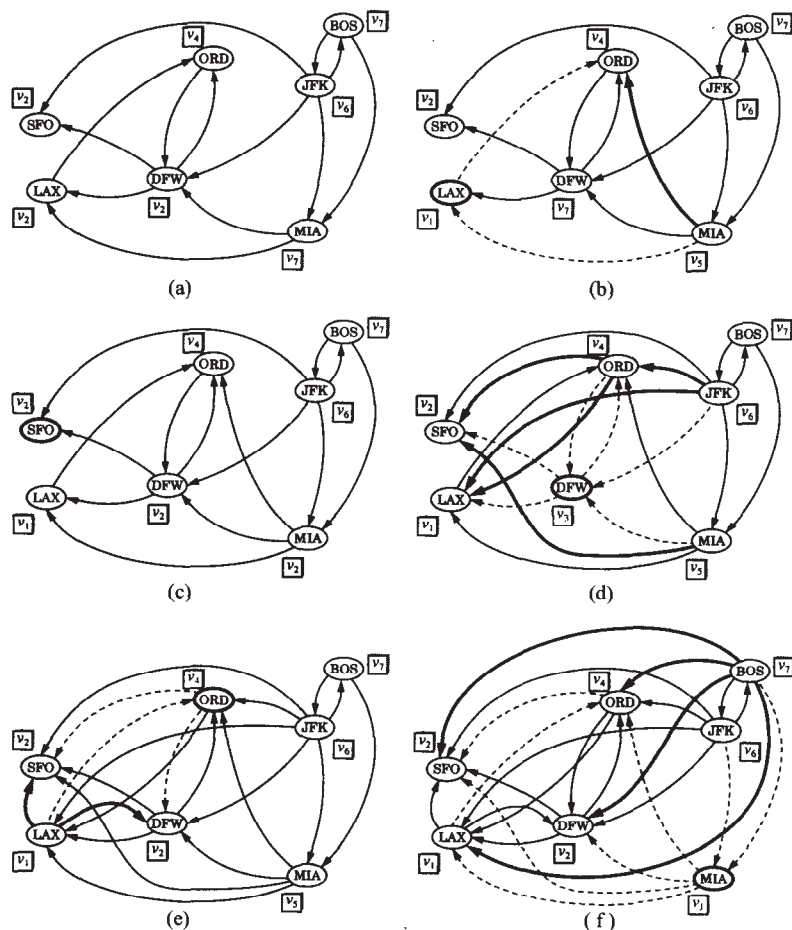


图6-13 Floyd-Warshall算法计算的有向图序列：(a)初始有向图 $\vec{G} = \vec{G}_0$ 和顶点的编号；(b)有向图 \vec{G}_1 ；(c)有向图 \vec{G}_2 ；(d) \vec{G}_3 ；(e) \vec{G}_4 ；(f) \vec{G}_5 。注意 $\vec{G}_5 = \vec{G}_6 = \vec{G}_7$ 。如果有向图 \vec{G}_{k-1} 中包含边 (v_i, v_k) 和 (v_k, v_j) ，但不含边 (v_i, v_j) ，则在绘制有向图 \vec{G}_k 时，用细虚线表示 (v_i, v_k) 和 (v_k, v_j) ，用粗实线表示 (v_i, v_j) 。

322

6.4.3 DFS 和垃圾收集

在某些语言（如C和C++）中，对象的内存空间必须由程序员显式分配和释放。初级程序员常常忽视这种内存分配的责任。甚至对于有经验的程序员，不正确的操作也是程序出现错误的来源。因此，其他语言（如Java）的设计者把内存管理的任务放在运行时环境中。当某个对象的生命周期终止时，Java程序员不必显式释放这些对象的内存空间。作为替代，垃圾收集器（garbage collector）机制会为这些对象释放内存空间。

在Java中，为大多数对象分配存储空间是由称为“内存堆”（不要与堆数据结构混淆）的内存池来完成的。此外，运行线程（2.1.2节）把实例变量的空间存储在它们各自的方法栈（2.1.1节）中。因为方法栈中的实例变量可以引用内存堆中的对象，因而，称运行线程方法栈中的所有变量和对象为根对象（root object）。所有这些对象都可从称为活动对象（live object）的根对象开始，沿着对象引用到达。活动对象就是运行程序当前使用的那些活动对象；这些对象不会被释放。例如，一个运行的Java程序可能在一个变量中存储一个指向双链表实现的序列S的引用。S的引用变量是一个根对象，而S的对象是一个活动对象，由此对象引用所有结点对象，并由这些结点对象引用所有的元素。

有时，Java虚拟机也会注意到内存堆中的可用空间变得稀少。这时，JVM可以选择回收那些被不再活动的对象使用的空间。称这个回收过程为垃圾收集（garbage collection）。有几个不同的垃圾收集算法，其中最常用的是标记-扫描（mark-sweep）算法。

1. 标记-扫描算法

在标记-扫描垃圾收集算法中，每个对象关联一个“标记”位，用以标识那个对象是否是活动的。当确定需要进行垃圾收集时，挂起所有其他运行的线程，并清除内存堆中当前分配的所有对象的标记位。然后通过当前运行线程的Java栈进行跟踪，并把这些栈中的所有（根）对象标记为“活动”状态。然后，确定所有其他活动对象——由根对象可达的那些对象。为使这个过程高效进行，应该利用有向图的深度优先查找算法。在这种情况下，内存堆中的每个对象均可被看作有向图中的一个顶点，从一个对象到另一个对象的引用则可被看作一条边。从每个根对象执行有向DFS过程，可以正确地确定并标记每个活动对象。称这个过程为“标记”阶段。一旦这个过程完成，就开始扫描内存堆，并回收那些未被标记的对象使用的空间。称这个扫描过程为“扫描”阶段。当它完成时，恢复运行挂起的线程。因此，标记-扫描垃圾收集算法将会回收不用的空间，其运行时间与活动对象数和引用数以及内存堆的大小成正比。

323

2. 原位执行DFS

标记-扫描算法正确地回收内存堆中不用的空间，但在标记阶段中，要面临一个重要的问题。因为当可用内存空间稀少时，才回收内存空间，所以必须注意在垃圾收集过程中，它自身不占用额外的空间。问题是采用递归方式描述DFS算法，该算法可以使用的空间与图中的顶点数成正比。在垃圾收集的情况下，图中的顶点就是内存堆中的对象；因此，没有更多内存空间可用。需要另一种方法执行原位DFS，而不是递归地执行。也就是说，必须只用额外的常量空间执行DFS过程。

执行原位DFS的主要思想是利用图的边（在垃圾收集的情况下，边对应于对象引用）模拟递归栈。每当从一个访问过的顶点v遍历一条边，并到达一个新的顶点时，就把存储在v的邻接表中的(v, w)改为指回DFS树中v的父顶点。当返回到v（从w的“递归”调用模拟返回）时，把修改后的边切换到指回w。当然，需要有一种方法，确定哪条边需要改回来。一种可能性是将出自于v的引用编号为1、2等并存储，除了标记位（在DFS中用于“访问过”的标签）之外，还要有一个计数标识符告诉我们哪些边被修改过。

利用计数标识符的过程对于每个对象要求额外一个字的存储空间。在某些实现中可以避免这个额外的字。例如,Java虚拟机的许多实现把对象表示为引用与类型标识符(用于表明这个对象是否是Integer类型或其他类型)的组合,以及表示为指向其他对象或者该对象的数据字段的引用。因为在这样的实现中,类型引用总是假定为组合中的第一个元素,当离开一个对象 v 并转到其他对象 w 时,可以利用这个引用标记改变的边。只需简单交换 v 处对 v 的类型的引用与 v 处对 w 的引用。当返回到 v 时,可以快速找出所改变的边 (v, w) ,因为它将是组合中第一个用于 v 的引用,并且指向 v 的类型的引用位置可以告知这条边在 v 的邻接表中的什么位置。因此,无论是利用这个交换边的技巧,还是利用计数标识符,都可以在原位实现DFS,而不会影响它的渐近运行时间。

324

6.4.4 有向无环图

在许多应用中可遇见有向图中没有有向回路的情况。这样的图常常称为有向无环图(directed acyclic graph),或简称为dag。这种图的应用包括:

- C++类或Java接口之间的继承。
- 学位课程之间的先修课程。
- 工程任务之间的调度。

示例6.8 为了管理大型工程项目,经常把它分解为一些较小任务的集合。然而,这些任务几乎不独立,因为在这些任务之间,存在调度上的约束条件(例如,在建筑项目中,订购钉子的任务要在楼顶钉甲板任务之前完成)。显然,调度约束条件中不能出现循环,因为循环使工程不可能完工(例如,为了得到一份工作,你需要有工作经验,但为了有工作经验,你需要有工作)。调度约束条件在要执行的任务之间强加了一个次序限制。也就是说,如果一个约束条件要求任务 a 必须在任务 b 开始之前完成,那么在任务的执行次序中,任务 a 必定在任务 b 之前。因此,如果把任务的一个可行集合建模为有向图中的顶点,并且无论何时 v 的任务必须在 w 的任务之前完成,都放置一条从 v 到 w 的有向边,这样,就定义了一个有向无环图。

上述例子激励了以下定义的产生。设 \vec{G} 是具有 n 个顶点的有向图。 \vec{G} 的一个拓扑序列(topological ordering)是 \vec{G} 中顶点的一个序列 (v_1, v_2, \dots, v_n) ,满足对于 \vec{G} 中的每条边 (v_i, v_j) ,都有 $i < j$ 。也就是说,一个拓扑序列是一种次序,满足 \vec{G} 中的任何有向路径均以增序遍历顶点(如图6-14所示)。注意,有向图中可能有多个拓扑序列。

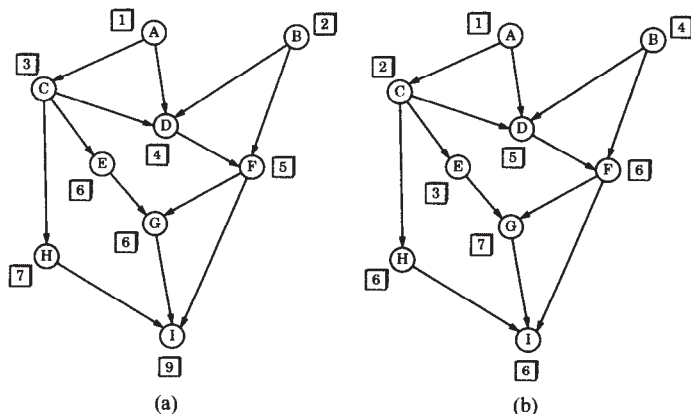


图6-14 同一个无环有向图中的两个拓扑序列

325

定理6.13 当且仅当有向图是无环的时，它才具有拓扑序列。

证明 必要性（声明的“仅当”部分）易于证明。假定 \vec{G} 中存在拓扑序列。用反证法证明，假设 \vec{G} 中存在一个由边 $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \dots, (v_{i_{k-1}}, v_{i_0})$ 组成的回路。由于拓扑序列，必定有 $i_0 < i_1 < \dots < i_{k-1} < i_0$ ，这显然是不可能的。因此， \vec{G} 必定是无环的。

现在证明充分性（“当”部分）。假定 \vec{G} 是无环的。我们描述一个算法，构建 \vec{G} 的一个拓扑序列。因为 \vec{G} 是无环的， \vec{G} 中必定存在一个顶点没有入边（即入度为0）。设 v_1 是这样的顶点。实际上，如果 v_1 不存在，那么从任一顶点开始跟踪一条有向路径，最终会遇见一个以前访问过的顶点，这与 \vec{G} 是无环图相矛盾。如果从 \vec{G} 中删除 v_1 以及它的出边，得到的有向图仍然是无环的。因此，得到的有向图中也存在一个顶点没有入边，设 v_2 是这样的顶点。重复这个过程，直到 \vec{G} 变为空图，此时得到 \vec{G} 中顶点的一个序列 v_1, v_2, \dots, v_n 。由上述构造过程，如果 (v_i, v_j) 是 \vec{G} 的一条边，那么， v_i 的删除一定发生在 v_j 可以被删除之前，因而 $i < j$ 。因此， v_1, v_2, \dots, v_n 是一个拓扑序列。 ■

上述证明过程表现在算法6-5中，称该算法为拓扑排序（topological sorting）算法。

算法6-5 拓扑排序算法

```

算法 TopologicalSort( $\vec{G}$ ):
  输入: 具有  $n$  个顶点的有向图  $\vec{G}$ 
  输出:  $\vec{G}$  的一个拓扑序列  $v_1, v_2, \dots, v_n$  或一个表明  $\vec{G}$  中存在有向回路的指示
  设  $S$  是初始为空的栈
  for  $\vec{G}$  中的每个顶点  $u$  do
     $\text{incounter}(u) \leftarrow \text{indeg}(u)$ 
    if  $\text{incounter}(u) = 0$  then
       $S.\text{push}(u)$ 
   $i \leftarrow 1$ 
  while  $S$  非空 do
     $u \leftarrow S.\text{pop}()$ 
    把  $u$  编号为第  $i$  个顶点  $v_i$ 
     $i \leftarrow i + 1$ 
    for 每条边  $e \in \vec{G}.\text{outIncidentEdges}(u)$  do
       $w \leftarrow \vec{G}.\text{opposite}(u, e)$ 
       $\text{incounter}(w) \leftarrow \text{incounter}(w) - 1$ 
      if  $\text{incounter}(w) = 0$  then
         $S.\text{push}(w)$ 
  if  $i > n$  then
    return  $v_1, \dots, v_n$ 
  else
    return “有向图  $\vec{G}$  中存在有向回路”
  
```

326

定理6.14 设 \vec{G} 是具有 n 个顶点、 m 条边的有向图。拓扑排序算法利用 $O(n)$ 的辅助空间，计算 \vec{G} 的一个拓扑序列，或者未能对某些顶点编号，这表明 \vec{G} 中存在有向回路。这个算法的运行时间为 $O(n + m)$ 。

证明 首先, 对图进行简单遍历, 计算顶点的入度并设置计数器变量, 所需时间为 $O(n+m)$ 。利用图结点中的另外一个域, 或者利用下一节中描述的修饰模式, 关联顶点的计数器属性。当从栈 S 中删除 u 时, 称这个顶点 u 被拓扑排序算法访问过 (visited)。仅当 $incounter(u) = 0$ 时, 顶点 u 才会被访问。这蕴涵着它的所有前驱 (出边进入 u 的顶点) 都已被访问过。因而, 有向回路上的任何顶点都不会被访问, 其他顶点则只会被访问一次。该算法会遍历每个被访问过的顶点的所有出边一次, 因此, 它的运行时间与所访问顶点的出边数成正比。于是, 算法的运行时间为 $O(n+m)$ 。至于占用空间, 观察可知, 栈 S 和附着在顶点上的计数器变量所用空间为 $O(n)$ 。

该算法的另一个作用是测试输入有向图 \vec{G} 是否是无环的。实际上, 如果算法终止, 且没有对所有顶点排序, 那么那些没有排序的顶点组成的子图一定包含有向回路 (如图6-15所示)。■

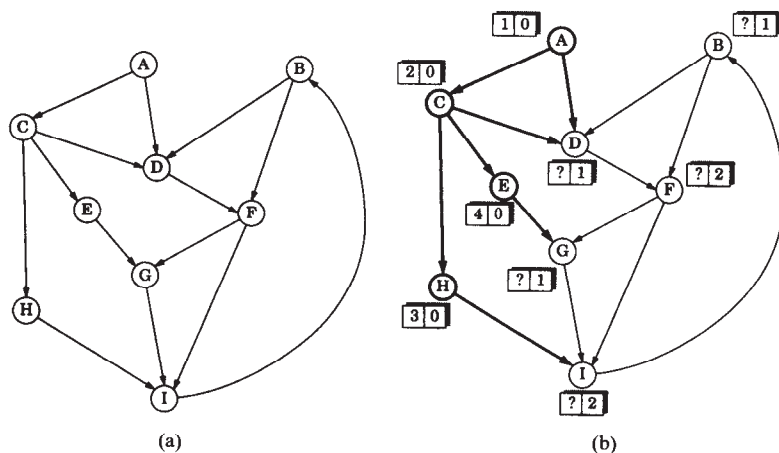


图6-15 检测一个有向回路: (a)输入有向图; (b)执行算法TopologicalSort (算法6-5) 之后, 未定义编号的顶点组成子图中包含有向回路

图6-16是拓扑排序算法的可视化过程。

327

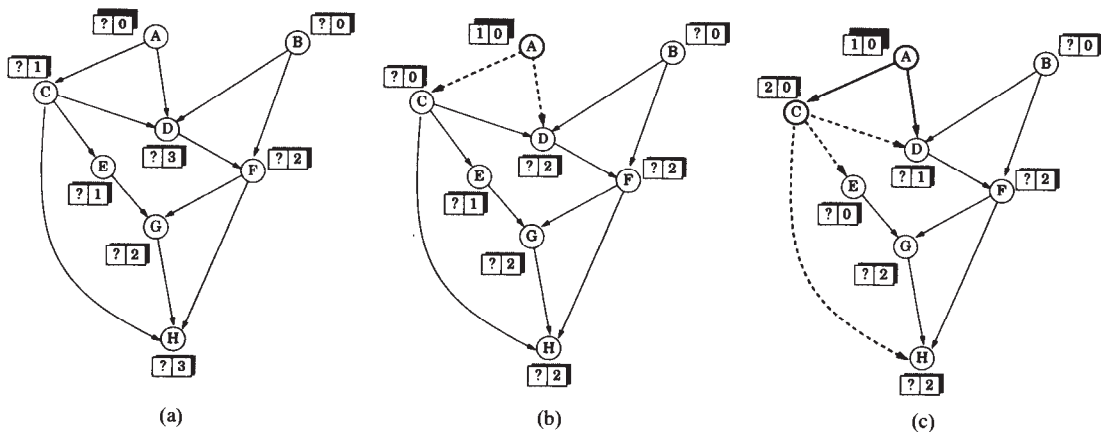


图6-16 拓扑排序算法TopologicalSort (算法6-5) 的运行示例: (a)初始配置; (b~i)每个while循环迭代之后。顶点标记给出顶点编号和当前计数器值。以前的迭代中遍历的边用粗实线表示。当前迭代中遍历的边用粗虚线表示

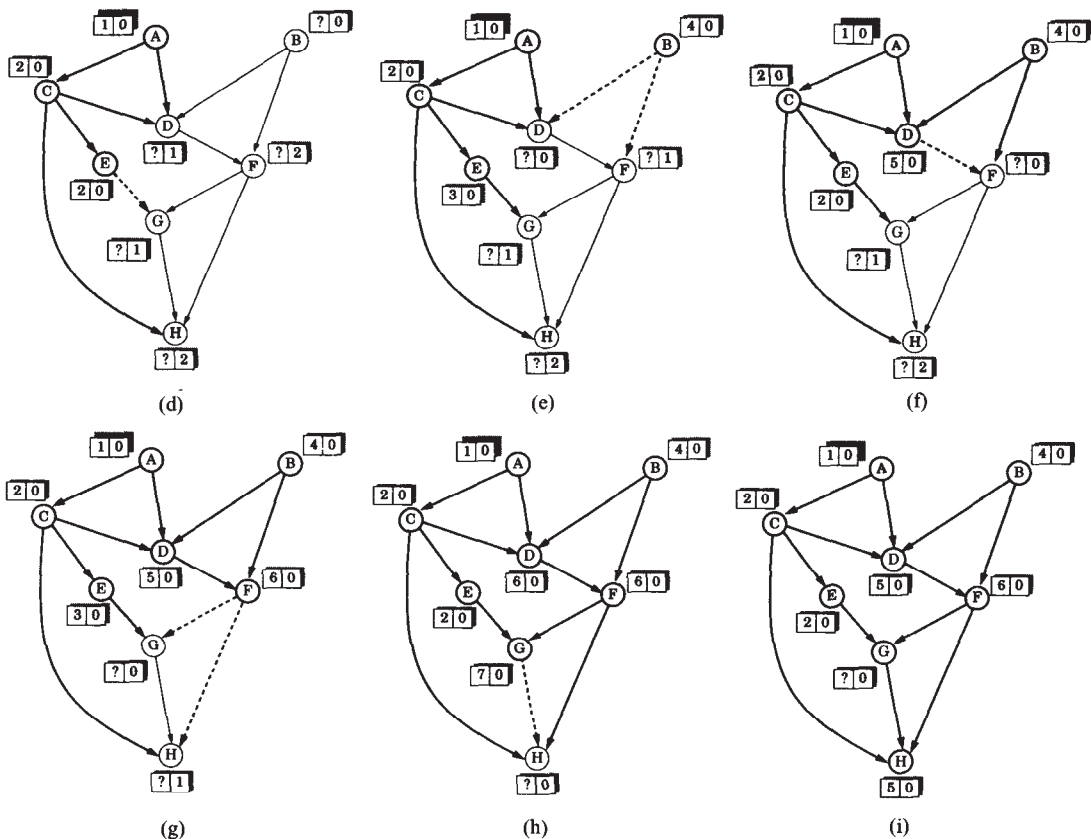


图6-16 (续)

328

6.5 Java 示例：深度优先查找

在这一节里，我们描述一个案例研究，讨论用Java实现深度优先查找算法。这个实现不是简单的Java伪代码的实例化，而是利用了两个有趣的软件工程设计模式——修饰模式和模板方法模式。

6.5.1 修饰模式

标记DFS遍历中探索过的顶点是修饰（decorator）软件工程设计模式的例子。修饰模式用于对现有对象添加属性或者“修饰”。每个属性用一个特定的修饰对象 a 标识，作为标识这个属性的一种关键字。直觉上，属性 a 是修饰的“名字”，我们允许这个属性对于不同的对象具有不同的值，而这些对象可能用 a 个属性修饰。修饰的使用是由于某些算法和数据结构中的对象需要增加额外的变量或者临时草稿数据，而通常没有这些变量。因此，修饰是一个数对(属性，值)，它能够动态地附着在一个对象上。例如，AVL树或红黑树中的结点可用高度或颜色属性进行修饰。在DFS例子中，希望对有些顶点和边用探索过的（explored）属性和一个布尔值进行修饰。

对于所有位置容器，可以向位置ADT中添加以下方法，来实现修饰模式。

- $\text{has}(a)$: 测试位置是否有属性 a 。

- `get(a)`: 返回属性 a 的值。
- `set(a, x)`: 将属性 a 的值设置为 x 。
- `destroy(a)`: 删除属性 a 及其关联的值。

实现修饰模式

例如, 在每个位置上, 把属性字典及其所属位置的值存储在一张小散列表中, 可以实现上述方法。也就是说, 属性可以是特定属性对象, 甚至是定义了属性“名字”的字符串。这些属性定义了散列表中使用的关键字。同样, 这些值是简单的基本类型, 或是把某些特定数据赋予给定属性的对象。这样的散列表实现提供了一种快速、有效的方式, 用于实现修饰模式中的方法。实际上, 尽管未加以说明, 但在DFS应用中, 利用一个小散列表实现修饰模式。

329

6.5.2 DFS引擎

在代码段6-1中, 给出了递归深度优先查找遍历的主Java“引擎”, 它包含在名为“DFS”的类中。

代码段6-1 DFS类中的主要变量和引擎 (dfsTraversal)

```

/** Generic depth first search traversal of a graph using the template
 * method pattern. A subclass should override various methods to add
 * functionality to this traversal. */
public abstract class DFS {
    /** The graph being traversed. */
    protected InspectableGraph G;
    /** The result of the traversal. */
    protected Object visitResult;
    /** Perform a depth first search traversal.
     * @param g Input graph.
     * @param start Start vertex of the traversal.
     * @param info Auxiliary information (to be used by subclasses).
     */
    public Object execute(InspectableGraph g, Vertex start, Object info) {
        G = g;
        for (PositionIterator pos = G.positions(); pos.hasNext(); )
            unVisit(pos.nextPosition());
        return null;
    }
    /**
     * Recursive template method for a generic DFS traversal.
     * @param v Start vertex of the traversal.
     */
    protected Object dfsTraversal(Vertex v) {
        initResult();
        startVisit(v);
        visit(v);
        for (EdgeIterator inEdges = G.incidentEdges(v); inEdges.hasNext(); ) {
            Edge nextEdge = inEdges.nextEdge();
            if (!isVisited(nextEdge)) { // found an unexplored edge, explore it
                visit(nextEdge);
                Vertex w = G.opposite(v, nextEdge);
                if (!isVisited(w)) { // w is unexplored, this is a discovery edge
                    traverseDiscovery(nextEdge, v);
                    if (!isDone())
                        visitResult = dfsTraversal(w);
                }
            }
        }
    }
}

```

```

    }
    else // w is explored, this is a back edge
        traverseBack(nextEdge, v);
    }
}
finishVisit(v);
return result();
}

```

330

6.5.3 模板方法设计模式

DFS类被定义成使得它的行为可以专用于任何特定的应用问题。在代码段6-2和代码段6-3中，给出了DFS类中执行特殊化行为和修饰行为的方法。其特殊化是通过重新定义方法execute以及下列方法做到的，execute方法的作用是激活计算，下列方法则会被递归模板方法dfsTraversal在多个时间调用。

- `initResult()`: 在dfsTraversal开始执行时调用该方法。
- `startVisit(Vertex v)`: 也在dfsTraversal开始执行时调用该方法。
- `traverseDiscovery(Edge e, Vertex v)`: 当遍历出自v的一条发现边e时，调用该方法。
- `traverseBack(Edge e, Vertex v)`: 当遍历出自v的一条后边e时，调用该方法。
- `isDone()`: 调用该方法，确定遍历是否过早结束。
- `finishVisit(Vertex v)`: 当遍历完v的所有依附边时，调用该方法。
- `result()`: 调用该方法以返回到dfsTraversal的输出。

代码段6-2 特殊化DFS类（代码段6-1）中的模板方法dfsTraversal使用的辅助方法

```

/** Auxiliary methods for specializing a generic DFS */
protected void initResult() {} // Initializes result (called first)
protected void startVisit(Vertex v) {} // Called when we first visit v
protected void finishVisit(Vertex v) {} // Called when we finish with v
protected void traverseDiscovery(Edge e, Vertex from) {} // Discovery edge
protected void traverseBack(Edge e, Vertex from) {} // Back edge
protected boolean isDone() { return false; } // Is DFS done early?
protected Object result() { return new Object(); } // The result of the DFS

```

代码段6-3 DFS类（代码段6-1）中的方法visit、unVisit和isVisited，用可修饰位置实现

```

/** Attribute and its two values for the visit status of positions. */
protected static Object STATUS = new Object(); // The status attribute
protected static Object VISITED = new Object(); // Visited value
protected static Object UNVISITED = new Object(); // Unvisited value
/** Mark a position as visited. */
protected void visit(Position p) { p.set(STATUS, VISITED); }
/** Mark a position as unvisited. */
protected void unVisit(Position p) { p.set(STATUS, UNVISITED); }
/** Test if a position has been visited. */
protected boolean isVisited(Position p) { return (p.get(STATUS) == VISITED); }

```

331

1. 模板方法模式定义

泛型深度优先查找遍历基于模板方法模式。它描述了一种泛型计算机制，可通过重新定义某些步骤来进行特殊化，即定义泛型类Algorithm，执行某些有用和可能复杂的函数，同时还在某些

点上调用命名的过程集合。初始时，过程不做任何事情，但通过扩展Algorithm类，重新定义这些方法，使之做一些有趣的事情，那么就能构造一个有价值的计算。Java中的Applet类就是使用这种设计模式的一个例子。在DFS应用中，利用这种模板方法模式进行深度优先查找，它假设底层图是无向图，但稍加修改即可适合于有向图。

在遍历中，区分顶点和边已被访问过的方式封装在方法isVisited、visit和unVisit的调用中。实现中（见代码段6-3）假设顶点和边的位置支持修饰模式，以下将要对其进行讨论（另外，在这种模式中，可以建立一个位置的字典，将访问过的顶点和边存储在其中）。

2. 利用DFS模板

要用dfsTraversal做任何有趣的事情，就必须扩展DFS类，并重新定义代码段6-2中的一些辅助方法，以执行一些有价值的任务。这种方法与模板方法模式一致，因为这些方法对模板方法dfsTraversal的行为进行了特殊化。

例如，代码段6-4中给出的类ConnectivityTesterDFS扩展了DFS，建立一个程序用于测试图是否是连通的。它对从某个顶点开始的DFS遍历中可达的顶点进行计数，并与图中的顶点总数进行比较。

代码段6-4 特殊化DFS类，用于测试图是否是连通的

```
/** This class specializes DFS to determine whether the graph is connected. */
public class ConnectivityTesterDFS extends DFS {
    protected int reached;
    public Object execute(InspectableGraph g, Vertex start, Object info) {
        super.execute(g, start, info);
        reached = 0;
        if (!G.isEmpty()) {
            Vertex v = G.aVertex();
            dfsTraversal(v);
        }
        return (new Boolean(reached == G.numVertices()));
    }
    public void startVisit(Vertex v) { reached++; }
}
```

332

3. 扩展DFS进行路径查找

可以通过更聪明的方式扩展DFS，使其执行更多有趣的算法。

例如，代码段6-5中给出的类FindPathDFS用于找出一条给定起始顶点和目的顶点之间的路径。它开始于某个顶点，进行深度优先查找遍历。在查找过程中，维持从起始顶点到当前顶点的发现边的路径。当遇见一个未探索过的顶点时，把它添加到路径的末尾，当处理完一个顶点时，则从路径中删除它。当遇见目的点时，遍历过程终止，并把路径作为顶点的迭代器返回。注意这个类找到的路径由发现边组成。

代码段6-5 特殊化DFS类，用于找出起始顶点和目的顶点之间的一条路径。辅助类VertexIteratorAdapter提供了一个从序列到顶点迭代器的适配器

```
/** This class specializes DFS to find a path between the start vertex
 * and a given target vertex. */
public class FindPathDFS extends DFS {
    protected Sequence path;
    protected boolean done;
```



```

protected Vertex target;
/** @param info target vertex of the path
 * @return Iterator of the vertices in a path from the start vertex
 * to the target vertex, or an empty iterator if no such path
 * exists in the graph */
public Object execute(InspectableGraph g, Vertex start, Object info) {
    super.execute(g, start, info);
    path = new NodeSequence();
    done = false;
    target = (Vertex) info; // target vertex is stored in info parameter
    dfsTraversal(start);
    return new VertexIteratorAdapter(path.elements());
}
protected void startVisit(Vertex v) {
    path.insertLast(v);
    if (v == target)
        done = true;
}
protected void finishVisit(Vertex v) {
    if (!done)
        path.remove(path.last());
}
protected boolean isDone() {
    return done;
}
}

```

333

4. 扩展DFS用于回路查找

还可以扩展DFS，如同FindCycleDFS类中那样，在给定顶点v的连通分量中找出一个回路。代码段6-6给出了这个类。算法从v开始进行深度优先查找遍历，当找到一条后边时，算法终止。方法返回由找到的后边形成的回路（可能为空）的迭代器。

代码段6-6 特殊化DFS类，用于在起始顶点的连通分量中找出一个回路。辅助类EdgeIteratorAdapter提供了一个从序列到边迭代器的适配器

```

/** Specialize DFS to find a cycle in connected component of start vertex. */
public class FindCycleDFS extends DFS {
    protected Sequence cycle; // sequence of edges of the cycle
    protected boolean done;
    protected Vertex cycleStart, target;
    /** @return Iterator of edges of a cycle in the component of start vertex */
    public Object execute(InspectableGraph g, Vertex start, Object info) {
        super.execute(g, start, info);
        cycle = new NodeSequence();
        done = false;
        dfsTraversal(start);
        if (!cycle.isEmpty() && start != cycleStart) {
            PositionIterator pos = cycle.positions();
            while (pos.hasNext()) { // remove the edges from start to cycleStart
                Position p = pos.nextPosition();
                Edge e = (Edge) p.element();
                cycle.remove(p);
                if (g.areIncident(cycleStart, e)) break;
            }
        }
        return new EdgeIteratorAdapter(cycle.elements());
    }
}

```

```

    }
    protected void finishVisit(Vertex v) {
        if (!cycle.isEmpty() && (!done)) cycle.remove(cycle.last());
    }
    protected void traverseDiscovery(Edge e, Vertex from) {
        if (!done) cycle.insertLast(e);
    }
    protected void traverseBack(Edge e, Vertex from) {
        if (!done) {
            cycle.insertLast(e); // back edge e creates a cycle
            cycleStart = G.opposite(from, e);
            done = true;
        }
    }
    protected boolean isDone() { return done; }
}

```

334

6.6 习题

基础题

- R-6.1 画出一个有12个顶点、18条边和3个连通分量的无向图 G 。如果 G 有66条边，为什么不可能画出有3个连通分量的 G ？
- R-6.2 设 G 是有 n 个顶点、 m 条边的简单连通图。解释为什么 $O(\log m)$ 是 $O(\log n)$ 。
- R-6.3 画出一个有8个顶点、16条边的简单连通有向图，满足每个顶点的入度和出度均为2。证明存在一个（非简单）回路，它包括图中的所有边，即你可以按照它们各自的方向，遍历所有边一次（称这样的回路为欧拉路径（Euler tour））。
- R-6.4 Bob喜欢外国语言，计划调度学习以下9门课程：LA15、LA16、LA22、LA31、LA32、LA126、LA127、LA141和LA169。课程的先修关系为：
- LA15: 无
 - LA16: LA15
 - LA22: 无
 - LA31: LA15
 - LA32: LA16、LA31
 - LA126: LA22、LA32
 - LA127: LA16
 - LA141: LA22、LA16
 - LA169: LA32
- 找出一个课程序列，使Bob满足所有先修课程要求。
- R-6.5 假定用边表结构表示一个有 n 个顶点、 m 条边的图 G 。在这种情况下，为什么insertVertex方法的运行时间为 $O(1)$ ，而removeVertex方法的运行时间为 $O(m)$ ？
- R-6.6 设 G 是顶点编号为1~18的图，设每个顶点的邻接顶点由下表给出：

顶点	邻接顶点
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

假设在对 G 的一次遍历中，按照上表中列出的次序，返回给定顶点的邻接顶点。

- a. 画出 G 。
 b. 从顶点1开始, 给出按照DFS遍历访问的顶点次序。
 c. 从顶点1开始, 给出按照BFS遍历访问的顶点次序。

335

R-6.7 对于以下各种情况, 选择是利用邻接表结构, 还是利用邻接矩阵结构? 论证你的选择。

- a. 图有10 000个顶点、20 000条边, 尽可能占用少的空间非常重要。
 b. 图有10 000个顶点、20 000 000条边, 尽可能占用少的空间非常重要。
 c. 无论占用多大的空间, 都需要尽可能快地答复查询areAdjacent。

R-6.8 对于有 n 个顶点的简单图, 利用邻接矩阵结构表示, 解释为什么DFS遍历的运行时间为 $\Theta(n^2)$ 。

R-6.9 画出图6-2中显示的有向图的传递闭包。

R-6.10 计算图6-11d中实线表示的有向图的拓扑序列。

R-6.11 对于算法6-5中显示的拓扑排序算法, 能否利用队列代替栈作为辅助数据结构?

R-6.12 给出图6-6中所示的通过DFS遍历标记边的次序。

R-6.13 对于图6-10, 重做习题R-6.12, 说明BFS遍历过程。

R-6.14 对于图6-12, 重做习题R-6.12, 说明有向DFS遍历过程。

创新题

C-6.1 证明定理6.4。

C-6.2 详细描述一个 $O(n+m)$ 时间的算法, 计算一个有 n 个顶点的、 m 条边的无向图 G 的所有连通分量。

C-6.3 设 T 是一棵根为起始顶点的生成树, 它是通过深度优先查找一个连通无向图 G 而得。论证为什么对于不在 T 中的 G 的每条边, 从 T 中一个顶点到它的一个祖先顶点是一条后边 (back edge)。

提示: 假定这样的非树边是一条交叉边, 基于DFS访问这条边上端点的顺序进行论证。

C-6.4 假定希望利用边表结构表示有 n 个顶点的图 G , 假设利用集合 $\{0, 1, \dots, n-1\}$ 中的整数标识顶点。描述如何实现容器 E , 使其支持方法areAdjacent的 $O(\log n)$ 的时间性能。在这种情况下, 如何实现这个方法?

C-6.5 给出引理6.1的证明。

C-6.6 证明如果一个图 G 至少有三个顶点, 那么, 仅当它有一个孤立顶点时, 它才有一条孤立边。

C-6.7 给出引理6.3的证明。

336

C-6.8 给出证明算法LinkComponents (算法6-2) 正确性的细节。

C-6.9 Tamarindo大学和世界上的许多其他学校正在进行多媒体项目的合作。它们建立了一个计算机网络, 利用通信链路形成一棵自由树把这些学校连接起来。这些学校决定在其中的一所学校里, 安装一台文件服务器, 共享这些学校之间的数据。因为链路上传输时间由链路安装和同步支配, 数据传输的成本与所用的链路数成正比。因此, 希望为文件服务器选择一个“中心”位置。给定一棵自由树 T 和树中的结点 v , v 的离心率 (eccentricity) 是从 v 到 T 中任何其他结点的最长路径长度。称 T 中具有最小离心率的结点为 T 的中心 (center)。

a. 给定一棵有 n 个结点的自由树 T , 设计一个有效算法, 计算 T 的中心。

b. 中心是唯一的吗? 如果不是, 一棵自由树可以有多少个不同的中心?

C-6.10 利用单一队列代替层容器 L_0, L_1, \dots 作为辅助数据结构, 表明如何进行BFS遍历。

C-6.11 证明: 如果 T 是一棵为连通图 G 产生的BFS树, 那么对于第 i 层的每个顶点 v , T 中 s 和 v 之间的路径有 i 条边, G 中 s 和 v 之间的任何其他路径至少有 i 条边。

C-6.12 在电话网络上的呼叫者和被叫者之间, 存在长距离呼叫延迟, 可通过在通话之间的通信链路数上乘以一个小的固定常数因子确定这个延迟。假定RT&T公司的电话网络是一棵自由树。RT&T公司的工程师想要计算在长距离通话中可能经历的最大时间延迟。给定一棵自由树 T , T 的直径 (diameter) 就是 T 中两个结点之间的最长路径长度。给出一个有效算法, 计算 T 的直径。

C-6.13 RT&T公司有一个通过 m 条高速通信链路连接的 n 个交换站的网络。每个客户的电话直接连向他/她所在区域的一个站。RT&T公司的工程师研制了一种原型视频-电话系统, 可以使两个客户在通话中相互看到对方。然而, 在客户之间传输视频信号所用的链路数不能超过4。假定RT&T公司的网络用图

表示。设计一个有效算法，利用4条以下的链路，对于每个站计算能够到达的站集。

C-6.14 解释在为有向图构造的BFS树中，为什么不存在前向非树边。

C-6.15 给出一个详细的有向DFS遍历算法的伪代码描述。如何确定一条非树边是一条后边、前向边，还是交叉边？

C-6.16 解释为什么6.4.1节给出的强连通性测试算法是正确的。

C-6.17 设 G 是有 n 个顶点、 m 条边的无向图。描述一个运行时间为 $O(n+m)$ 的算法，在每个方向上遍历 G 中的每条边一次。

337

C-6.18 无向图 $G=(V, E)$ 的一个独立集是 V 的一个子集 I ，满足 I 中不存在任意两个顶点是相邻的。也就是说，如果 $u, v \in I$ ，那么， $(u, v) \notin E$ 。最大独立集 M 是一个独立集，满足如果向 M 中增加任何额外的顶点，它就不再是独立集。每个图都有一个最大独立集（你能够看出这一点吗？这个问题不是本习题的一部分，但值得思考）。给出一个有效算法，计算图 G 的一个最大独立集。算法的运行时间是多少？

C-6.19 具有 n 个顶点、 m 条边的有向图 \vec{G} 的欧拉路径（Euler tour）是一个回路，它按照 \vec{G} 中边的方向遍历其中的每条边一次。如果 \vec{G} 是连通的，且 \vec{G} 中每个顶点的入度等于出度，那么这样的路径总是存在。描述一个 $O(n+m)$ 时间的算法，找出这样一个有向图 \vec{G} 的欧拉路径。

C-6.20 证明定理6.8。

程序设计

P-6.1 实现一个简化的图ADT，它利用邻接矩阵结构，只包含与无向图有关的方法，并且不包含更新方法。

P-6.2 利用邻接表结构，实现P-6.1中描述的简化图ADT。

P-6.3 利用模板方法模式，实现泛型BFS遍历。

P-6.4 实现拓扑排序算法。

P-6.5 实现Floyd-Warshall传递闭包算法。

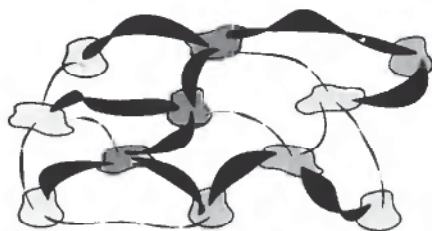
6.7 本章笔记

深度优先查找方法是计算机科学悠久传统的一部分，但是Hopcroft和Tarjan[98, 198]利用这个算法解决了图论中的几个不同的问题。Knuth[118]讨论了拓扑排序问题。6.4.1节描述的确定一个有向图是否是强连通的简单线性时间算法由Kosaraju提出。Floyd-Warshall算法出现在Floyd[69]的一篇论文中，并基于Warshall[209]提出的一个定理。本章描述的标记-扫描垃圾收集方法是许多不同的垃圾收集算法之一。我们鼓励对垃圾收集的研究感兴趣的读者进一步阅读Jones[110]的著作。要学习不同的画图算法，请参考Tamassia[194]的著作以及Di Battista等人[58]的注释文献，或者Di Battista等人[59]的著作。

对图论算法感兴趣的读者，可进一步参阅Ahuja、Magnanti和Orlin[9]，Cormen、Leiserson和Rivest[55]、Even[68]、Gibbons[77]、Mehlhorn[149]、Tarjan[200]以及van Leeuwen[205]的著作。

338

第7章



加权图

正如在上一章中所看到的，广度优先查找策略可用于找出一个连通图中从某个起始顶点到每个其他顶点的最短路径。这种方法在边比较均等时有意义，而在其他许多情况下这种方法并不合适。

例如，可以利用图表示一个计算机网络（如因特网），可以找出一种最快的方式在两台计算机之间路由数据包。在这种情况下，所有边彼此相等可能不太合适，因为计算机网络中的某些连接通常比其他一些连接快得多（例如，某些边可能表示慢速电话线的连接，而其他一些边可能表示高速、光纤连接）。同样，我们想要利用图表示城市之间的道路，以及对找出穿越国家之间的最快方式感兴趣。在这种情况下，所有边彼此相等也不太合适，因为某些城市间的距离可能比另外一些城市间的距离更大。因此，自然要考虑图中边的权值不相等。

在这一章里，研究加权图。加权图（weighted graph）是一个图，图中每条边 e 上关联一个数值 $w(e)$ ，称为边 e 的权值（weight）。边的权值可以为整数、有理数或实数，它可以表示诸如距离、连接成本或亲缘关系之类的概念。加权图的一个例子如图7-1所示。

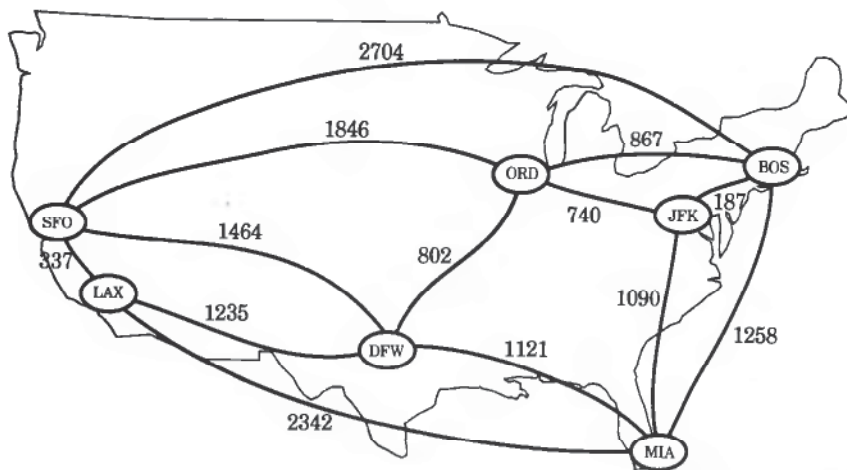


图7-1 加权图，其顶点表示美国主要的机场，边上的权值表示以英里计算的距离。这个图中从JFK到LAX的路径上的总权值为2777（经过ORD和DFW）。这是图中从JFK到LAX的最短加权路径

7.1 单源点最短路径

设 G 是加权图。路径 P 上的长度 (length) 或权值 (weight) 是 P 上边的所有权值之和。也就是说, 如果 P 由边 e_0, e_1, \dots, e_{k-1} 组成, 那么, P 的长度 (表示为 $w(P)$) 定义如下

$$w(P) = \sum_{i=0}^{k-1} w(e_i)$$

G 中如果存在从顶点 v 到顶点 u 的路径, 则这条路径上的距离 (distance) 为从 v 到 u 的最短长度路径 (也称最短路径 (shortest path)) 的长度, 用 $d(v, u)$ 表示。

如果 G 中从 v 到 u 根本不存在路径, 则人们常常使用约定 $d(v, u) = +\infty$ 。即使从 v 到 u 存在路径, 但从 v 到 u 的距离可能没有定义, 这是因为如果 G 中存在一个其总权值为负的回路。例如, 假定 G 中的顶点表示城市, 边表示从一个城市到另一个城市所需的成本。如果某个人想要实际地支付从JFK到ORD的成本, 那么边(JFK, ORD)的“成本”为负。如果另一个人想要支付从ORD到JFK的成本, 那么 G 中存在具有负权值的回路, 距离不再会被定义。也就是说, 任何人都可建立 G 中从任一城市 A 到另一城市 B 的路径 (带有回路), 这条路径在到达 B 之前, 首先会达到JFK, 然后经过多次从ORD到JFK并返回的回路。这样的路径存在, 使我们可以建立成本为任意小负值的路径 (在这种情况下, 在这个过程中可以赚大钱)。但是距离不能是任意小的负数。因此, 在利用边的权值表示距离时, 必须小心不要引入任何具有负权值的回路。

给定一个加权图 G , 要求找出从某个顶点 v 到 G 中其他每个顶点的一条最短路径。把边上的权值看作距离。在这一节里, 如果这些路径存在, 探索找出所有这样的单源点最短路径 (single-source shortest path) 的有效方法。

讨论的第一个算法针对简单而常见的情况, 其中图 G 中的所有边的权值非负 (即对于 G 中的每条边 e , $w(e) \geq 0$); 因此, 预先知道 G 中不存在具有负权值的回路。回忆可知, 计算一条最短路径的特例是当所有权值为1时的情况, 可用6.3.3节给出的BFS遍历算法求解。

有一个有趣的方法用于求解这个单源点问题, 它基于贪心法 (greedy method) 设计模式 (5.1节)。回忆可知在这种模式中, 在每次迭代中通过从那些可用集合中反复选择当前最好的解来求解问题。这种范型常常试图用于优化对象集合上的某个成本函数。可以一次向集合中添加一个对象, 并且总是从那些要被选取的对象中选择下一个使函数最优的对象。

341

7.1.1 Dijkstra 算法

把贪心法模式应用于单源点最短路径问题的主要思想是, 以顶点 v 为起始顶点, 进行一个“加权”广度优先查找。尤其是, 利用贪心法开发一个算法, 反复生长出一个出自 v 的顶点“云集”, 这些顶点按照与 v 的距离的次序进入这个云集。因此, 在每次迭代中, 选择的下一个顶点是云集之外据 v 最近的顶点。当云集之外不存在顶点时, 算法终止。此时, 得到从 v 到 G 中每个其他顶点的最短路径。这个方法是简单的, 但却很强大, 它是贪心法设计模式的一个示例。

1. 用于找出最短路径的贪心法

把贪心法应用于单源点最短路径问题所得的算法称为Dijkstra算法 (Dijkstra algorithm)。但是, 当把贪心法应用于其他图论问题时, 这个方法不一定会找到问题的最优解 (如所谓的旅行推销员问题 (traveling salesman problem), 在这个问题中, 希望找到访问图中所有顶点恰好一次的最短路径)。然而, 有许多情况都可用贪心算法计算最优解。在这一章里, 讨论两种这样的情况: 计算最短路径和构造最小生成树。

为了简化Dijkstra算法的描述, 以下假设输入图 G 是无向(即它的所有边都是无向的)、简单(不存在自环和平行边)图。因此, 把图 G 的边表示成为无序的顶点对 (u, z) 。将应用于加权有向图的Dijkstra算法描述留作习题(R-7.2)。

在Dijkstra算法中, 试图应用贪心法优化的成本函数也是正尝试计算的一个函数——最短路径距离。这初看上去像是循环推理, 实际的实现是用“自展”的策略, 利用距离函数的一个近似函数进行计算, 最后, 这个函数值将等于真实的距离。

2. 边松弛

将 G 中顶点 v 的标记定义为 $D[u]$, 利用它表示 G 中从 v 到 u 的近似距离。这些标记的意义是, $D[u]$ 中总是存储迄今为止发现的从 v 到 u 的最佳距离的长度。起初, 对于每个 $u \neq v$, $D[v] = 0$, $D[u] = +\infty$, 定义集合 C , 它是我们顶点的“云集”, 初始为空集 \emptyset 。在算法的每次迭代过程中, 选择一个不在 C 中且具有最小 $D[u]$ 标记的顶点 u , 把 u 放入 C 中。当然, 在要进行的第一次迭代中, 把 v 放入 C 中。一旦把一个新顶点 u 放入 C 中, 就更新与 u 相邻且在 C 之外的每个顶点 z 的标记 $D[z]$, 以反映如下事实, 即可能存在经过 u 到达 z 的更好的新方式。这个更新操作称为松弛(relaxation)过程, 因为它取一个旧的估值, 并检查它是否能得到一个更好的估值(为什么称它为估值, 这隐含着伸展一个弹簧, 然后“放松”并回到其静止不动的状态)。在Dijkstra算法的例子, 对一条边 (u, z) 进行松弛, 满足已经计算了 $D[u]$ 的一个新值, 并希望检查这个新值是否是一条利用边 (u, z) 所得到的更好的值。这个特定的边松弛操作如下:

边松弛:

```
if  $D[u] + w(u, z) < D[z]$  then
     $D[z] \leftarrow D[u] + w(u, z)$ 
```

注意: 如果新发现的到达 u 的路径不比原来的路径好, 那么 $D[z]$ 将保持不变。

3. Dijkstra算法的细节

算法7-1中给出了Dijkstra算法的伪代码描述。注意利用优先队列 Q 存储云集 C 外的顶点。

算法7-1 Dijkstra算法用于计算图 G 的单源点最短路径问题, 其中起始顶点为 v

```
算法 DijkstraShortestPaths( $G, v$ ):
    输入: 具有非负权值的简单无向加权图 $G$ , 以及 $G$ 的一个特殊顶点 $v$ 
    输出: 对于 $G$ 中的每个顶点 $u$ , 输出标记 $D[u]$ , 满足 $D[u]$ 是 $G$ 中从 $v$ 到 $u$ 的距离
     $D[v] \leftarrow 0$ 
    for  $G$ 中的每个顶点 $u \neq v$  do
         $D[u] \leftarrow +\infty$ 
    设优先队列 $Q$ 包含 $G$ 中的所有顶点, 利用 $D$ 标记作为关键字
    while  $Q$ 非空 do
        {把一个新顶点 $u$ 加入云集}
         $u \leftarrow Q.removeMin()$ 
        for  $u$ 的每个邻接顶点 $z$ 且 $z$ 在 $Q$ 中 do
            {对边 $(u, z)$ 进行松弛过程}
            if  $D[u] + w(u, z) < D[z]$  then
                 $D[z] \leftarrow D[u] + w(u, z)$ 
                改变 $Q$ 中的顶点 $z$ 的关键字 $D[z]$ 
    return 每个顶点 $u$ 的标记 $D[u]$ 
```

图7-2和图7-3描述了Dijkstra算法的若干次迭代过程。

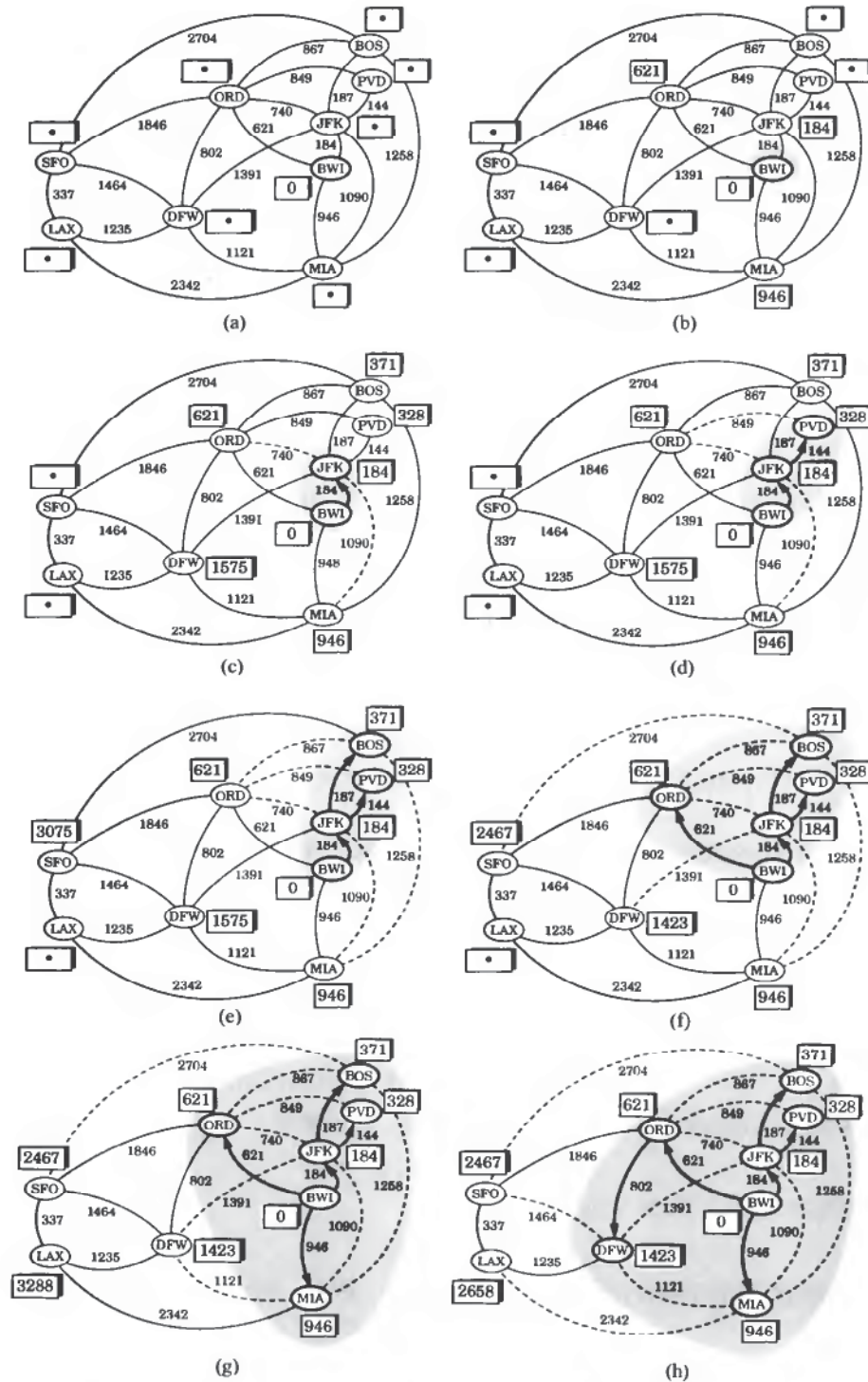


图7-2 在加权图上执行Dijkstra算法的过程。起始顶点为BWL。与每个顶点 u 紧邻的集合存储标记 $D[u]$ 。用符号 \bullet 代替 ∞ 。粗箭头表示最短路径树的边，对于“云集”外的每个顶点 u ，用粗线表示当前靠近 u 的最好边（图7-3续）

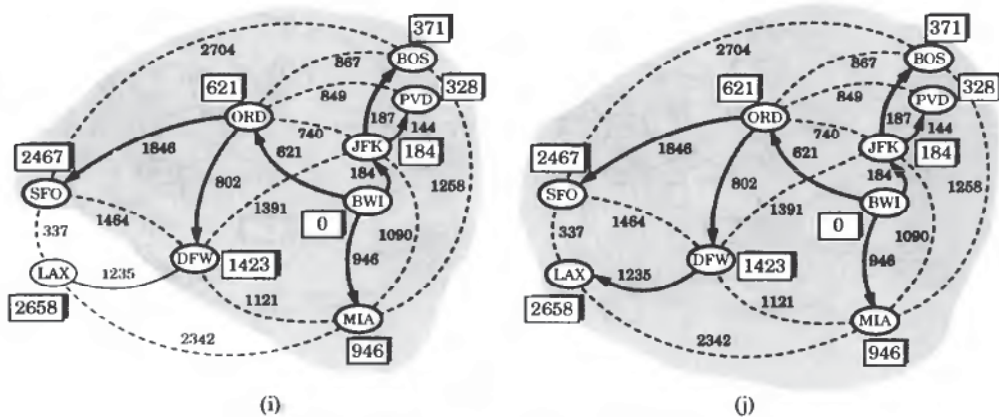


图7-3 Dijkstra算法的可视化 (图7-2续)

4. 为什么算法会工作

有趣甚至可能有点令人惊讶的是, 在Dijkstra算法中, 在把顶点 u 放入 C 中的那一刻, 把从 v 到 u 的最短路径的正确长度存储在其标记 $D[u]$ 中。因此, 当算法终止时, 已经计算了从 v 到 G 中每个顶点的最短路径的距离, 即已经解决了单源点最短路径问题。

Dijkstra算法为什么能够正确找出从起始顶点 v 到图中每个其他顶点 u 的最短距离, 其原因可能不是那么明显。为什么在顶点 u 被放入云集 C 中时 (此时将 u 从优先队列 Q 中删除), 从 v 到 u 的距离等于标记 $D[u]$ 的值? 这个问题的答案依赖于图中不存在具有负权值的边, 使得贪心法能够正确工作, 正如以下引理所示。

345

引理7.1 在Dijkstra算法中, 每当把一个顶点 u 放入云集中时, 标记 $D[u]$ 都等于从 v 到 u 的最短路径的长度 $d(v, u)$ 。

证明 对于 V 中的顶点 u , 假定 $D[u] > d(v, u)$ 。设 u 是算法放入云集 C 中的第一个顶点 (即从 Q 中删除), 满足 $D[u] > d(v, u)$ 。存在从 v 到 u 的最短路径 P (否则, $d(v, u) = +\infty = D[u]$)。于是, 考虑将 u 放入 C 中时的情况, 设 z 是路径 P 上的第一个顶点 (从 v 经过若干顶点到达 u), 且此时不在 C 中。 y 是路径 P 上 z 的前驱 (注意, 可能有 $y = v$) (如图7-4所示)。根据 z 的选择, 可知此时 y 已经在 C 中。此外, $D[y] = d(v, y)$, 因为 u 是第一个不正确的顶点。当将 y 放入 C 中时, 测试 $D[z]$ (并且可能更新 $D[z]$), 满足

$$D[z] \leq D[y] + w(y, z) = d(v, y) + w(y, z)$$

但是因为 z 是从 v 到 u 的最短路径上的下一个顶点, 这蕴涵着

$$D[z] = d(v, z)$$

但此时选择将 u 而不是 z 加入到 C 中; 因而,

$$D[u] \leq D[z]$$

显然, 最短路径的子路径自身也是一条最短路径。因此, 由 z 在从 v 到 u 的最短路径上可得

$$d(v, z) + d(z, u) = d(v, u)$$

此外, 由于不存在具有负权值的边, 所以 $d(z, u) \geq 0$ 。于是

$$D[u] \leq D[z] = d(v, z) \leq d(v, z) + d(z, u) = d(v, u)$$

但这与 u 的定义相矛盾; 因此, 不可能存在这样的顶点 u 。 ■

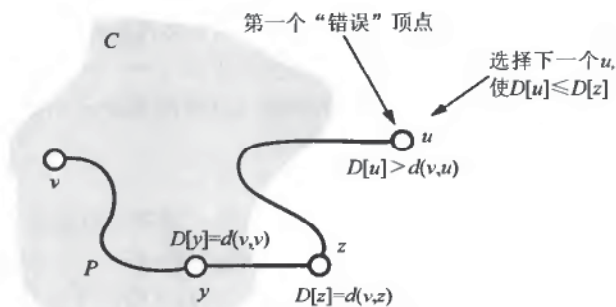


图7-4 引理7.1证明过程的示意图说明

346

5. Dijkstra算法的运行时间

在这一节里, 分析Dijkstra算法的时间复杂度。用 n 和 m 分别表示输入图 G 中的顶点数和边数。假设添加和比较边权值的时间为常数。由于算法7-1给出的是Dijkstra算法的高级描述, 分析它的运行时间需要给出其实现的更多细节。特别是, 应该指出使用的数据结构以及它们的实现方式。

首先假设利用邻接表结构表示图 G 。这种数据结构允许在松弛步骤中经过与 u 相邻的顶点的时间与其个数成正比。然而, 这仍然不能解决算法的细节, 因为我们必须更多地讨论如何实现算法中另一个主要的数据结构——优先队列 Q 。

利用堆(见2.4.3节)是一种有效实现优先队列 Q 的方法。通过调用removeMin方法, 允许提取具有最小标记 D 的顶点 u , 所用时间为 $O(\log n)$ 。正如伪代码所指出的, 每当更新 $D[z]$ 标记时, 都需要更新优先队列中 z 的关键字。例如, 如果用堆实现 Q , 那么更新这个关键字的方法是首先删除它, 然后插入带有新关键字的 z 。如果优先队列 Q 支持定位器模式(见2.4.4节), 那么就能容易地在 $O(\log n)$ 时间内实现这个关键字的更新操作, 因为顶点 z 的定位器可使 Q 立即访问堆中存储 z 的数据项(见2.4.4节)。假设利用堆实现优先队列 Q , 那么Dijkstra算法的运行时间为 $O((n+m) \log n)$ 。

回到算法7-1, 算法运行时间分析的细节如下:

- 反复向 Q 中插入带有初始关键字值的所有顶点, 所需时间为 $O(n \log n)$ 。如果利用自底向上构造堆(见2.4.4节), 则所需时间为 $O(n)$ 。
- 在while循环的每次迭代中, 从 Q 中删除顶点 u 所需时间为 $O(\log n)$, 对于依附于顶点 u 的边, 执行松弛过程所需的时间为 $O(\deg(v) \log n)$ 。
- while循环的总运行时间为

$$\sum_{v \in G} (1 + \deg(v)) \log n$$

由定理6.1可知, 其运行时间为 $O((n+m) \log n)$ 。

由此可得以下定理。

定理7.1 给定具有 n 个顶点、 m 条边的加权图 G , 以及每一条边上的非负权值, Dijkstra算法可实现或找出从顶点 v 开始到图 G 中其他所有顶点的最短路径, 其运行时间为 $O(m \log n)$ 。

注意, 如果希望将上述运行时间只表示为 n 的函数。那么在最坏情况下, 这个运行时间为 $O(n^2 \log n)$, 因为假设 G 是简单图。

347

6. Dijkstra算法的另一种实现

现在考虑用无序序列实现优先队列 Q 的另一种方法。当然, 假如 Q 支持定位器模式(见2.4.4节),

这种方法要求花费 $\Omega(n)$ 时间提取一个最小的元素,但它允许非常快速的关键字更新。确切地讲,在一个松弛步骤中,可用 $O(1)$ 时间实现每个关键字的更新——一旦定位到 Q 中要更新的数据项,则只须改变关键字值一次。因此,这种实现所导致的运行时间为 $O(n^2 + m)$,如果 G 是简单图,则可简化为 $O(n^2)$ 。

7. 比较两种实现

在Dijkstra算法中,有两种实现优先队列的方法:基于定位器的堆实现,得到 $O(m \log n)$ 的运行时间;基于定位器的无序序列实现,得到 $O(n^2)$ 的运行时间。因为这两种实现的编码相当简单,就所需的程序复杂性而言,这两种实现基本相同。就最坏情况下运行时间内所隐含的常数因子而言,这两种实现也基本相同。只看最坏情况下的运行时间,当图中的边数较少时(即当 $m < n^2/\log n$ 时),首选堆实现;而当图中的边数较大时(即当 $m > n^2/\log n$ 时),首选序列实现。

定理7.2 给定具有 n 个顶点、 m 条边的简单加权图 G ,以及 G 的一个顶点 v ,满足每一条边上的权值非负。Dijkstra算法计算从顶点 v 开始到图 G 中其他所有顶点的最短路径,运行时间为 $O(m \log n)$,或者为 $O(n^2)$ 。

在习题R-7.3中,探索如何修改Dijkstra算法,输出一棵根为 v 的树 T ,满足 T 中从 v 到顶点 u 的路径是 G 中从 v 到 u 的最短路径。此外,把Dijkstra算法扩展到用于有向图相当直观。但不能把Dijkstra算法扩展到用于带有负权值边的图。如图7-5中的说明。

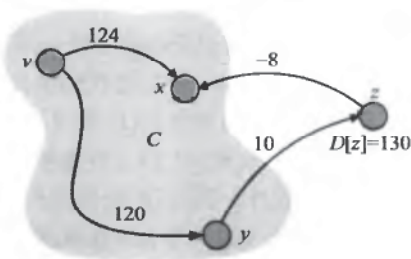


图7-5 为什么Dijkstra算法不能用于带有负权值边的图的说明。把 z 加入 C 中并进行边松弛,将使前面计算的到达 x 的最短路径距离(124)无效

348

7.1.2 Bellman-Ford 最短路径算法

Bellman和Ford提出了求带有负权值边的图中最短路径另一种算法。在这种情况下,必须强调图是有向的;否则,任意具有负权值的无向边可能直接蕴涵着一个具有负权值的回路,这会导致在每个方向上在这条边上来回遍历。不能允许这样的边存在,因为一个负回路会使基于边权值的距离概念无效。

设 \vec{G} 是加权有向图,可能含有一些具有负权值的边。Bellman-Ford算法用于计算从某个顶点 v 到 \vec{G} 中每个其他顶点的最短路径距离,它非常简单。该算法共享了Dijkstra算法中边松弛的概念,但它并不与贪心法一起使用(在这种环境中,贪心法不起作用,见习题C-7.2),即像在Dijkstra算法中一样,Bellman-Ford算法利用标记 $D[u]$,它总是从 v 到 u 的距离 $d(v, u)$ 的一个上界,这个上界反复被松弛,直到它正好等于这个距离为止。

Bellman-Ford算法的细节

算法7-2显示了Bellman-Ford算法。对于有向图,算法在每条边的松弛中会执行 $n-1$ 次。图7-6说明了Bellman-Ford算法的执行过程。

算法7-2 Bellman-Ford单源点最短路径算法，它允许图中包含具有负权值的边

算法 BellmanFordShortestPaths(\vec{G}, v):

输入: 具有 n 个顶点的加权有向图 \vec{G} ，以及 \vec{G} 中的一个顶点 v

输出: 对于 \vec{G} 中的每个顶点 u ，输出标记 $D[u]$ ，满足 $D[u]$ 是 \vec{G} 中从 v 到 u 的距离，或者表明 \vec{G} 中存在负权值回路的指示

$D[v] \leftarrow 0$

for \vec{G} 中每个顶点 $u \neq v$ do

$D[u] \leftarrow +\infty$

for $i \leftarrow 1$ to $n-1$ do

for 每条出自 u 的（有向）边 (u, z) do

{对 (u, z) 进行松弛操作}

if $D[u] + w((u, z)) < D[z]$ then

$D[z] \leftarrow D[u] + w((u, z))$

if 不存在余有潜在松弛操作的边 then

return 每个顶点 u 的标记 $D[u]$

else

return “ \vec{G} 包含负权值回路”

349

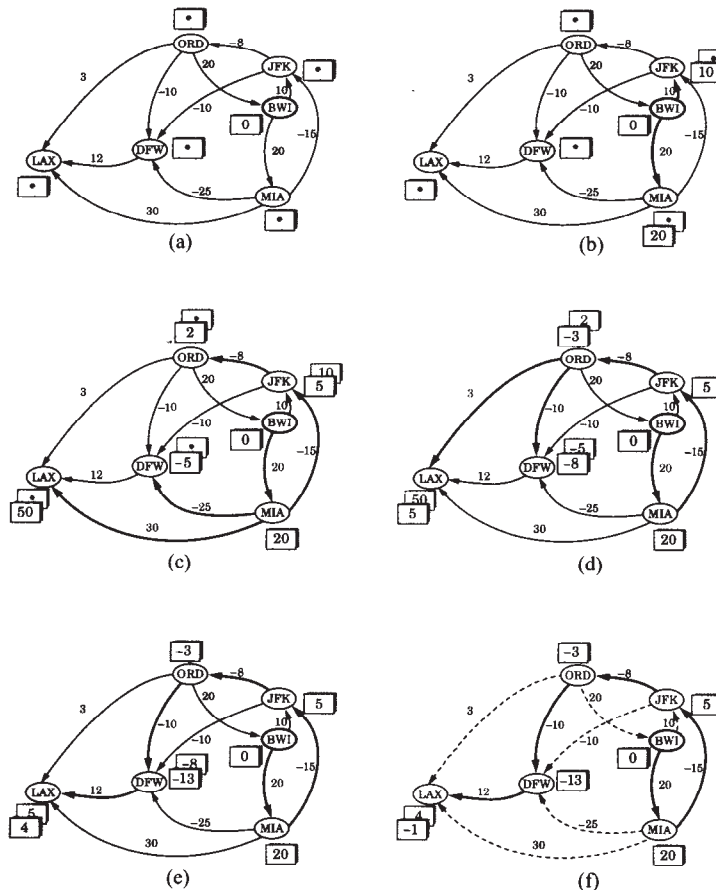


图7-6 Bellman-Ford算法的应用说明。起始顶点为BWI。每个顶点 u 的相邻顶点集存储标记 $D[u]$ ，用“阴影”表示松弛过程中修改过的值；粗边正在引起这样的松弛

350

引理7.2 如果算法7-2执行结束, 则存在一条边 (u, z) 可以被松弛 (即 $D[u] + w(u, z) < D[z]$), 那么输入有向图 \vec{G} 包含负权值回路。否则, 对于 \vec{G} 中的每个顶点 u , 有 $D[u] = d(v, u)$ 。

证明 为方便证明, 引入有向图中新的距离概念。设 $d_i(v, u)$ 表示从 v 到 u 的至多包含 i 条边的所有路径中最短路径的长度。声明对于 \vec{G} 中的每个顶点, 在Bellman-Ford算法的主for循环的第 i 次迭代后, $D[u] = d_i(v, u)$ 。在开始第一次迭代之前, 这肯定为真, 因为 $D[v] = 0 = d_0(v, v)$, 且对于 $u \neq v$, $D[u] = +\infty = d_0(v, u)$ 。假定第 i 次迭代之前, 这个声明为真 (将要证明: 如果是这种情况, 则在第 i 次迭代之后, 声明仍然为真)。在第 i 次迭代中, 对于有向图中的每条边, 执行松弛步骤。从 v 到顶点 u 的第 i 边的距离 $d_i(v, u)$ 由两种情况之一确定。对于 \vec{G} 中某个顶点 z , $d_i(v, u) = d_{i-1}(v, u)$, 或者 $d_i(v, u) = d_{i-1}(v, z) + w(z, u)$ 。因为在第 i 次迭代中, 对于 \vec{G} 中的每一条边进行松弛, 如果是前一种情况, 则在第 i 次迭代后, 有 $D[u] = d_{i-1}(v, u) = d_i(v, u)$; 如果是后一种情况, 则在第 i 次迭代后, 有 $D[u] = D[z] + w(z, u) = d_{i-1}(v, z) + w(z, u) = d_i(v, u)$ 。因此, 在第 i 次迭代之前, 对于每个顶点 u , 如果 $D[u] = d_{i-1}(v, u)$, 那么, 在第 i 次迭代之后, 对于每个顶点 u , $D[u] = d_i(v, u)$ 。

于是, $n-1$ 次迭代之后, 对于 \vec{G} 中每个顶点 u , 有 $D[u] = d_{n-1}(v, u)$ 。现在观察可见, 如果 \vec{G} 中仍然存在一条边可以被松弛, 那么, \vec{G} 中存在某个顶点 u , 满足从 v 到 u 的第 n 边的距离小于从 v 到 u 的第 $(n-1)$ 边的距离, 即 $d_n(v, u) < d_{n-1}(v, u)$ 。但是, \vec{G} 中只有 n 个顶点; 因此, 如果存在从 v 到 u 的最短第 n 边路径, 则必定是 \vec{G} 中某个顶点 z 重复了两次, 即必定包含回路。此外, 由于从某个顶点到其自身的距离利用了0条边, 距离为0 (即 $d_0(z, z) = 0$), 因此这个回路必定是负权值回路。因此, 如果 \vec{G} 中存在某条边在运行Bellman-Ford算法之后仍然能够被松弛, 那么 \vec{G} 中包含负权值回路。另一方面, 如果 \vec{G} 中不存在某条边在运行Bellman-Ford算法之后能够被松弛, 那么 \vec{G} 中不含负权值回路。此外, 在这种情况下, 两个顶点之间的每条最短路径至多有 $n-1$ 条边; 因此, 对于 \vec{G} 中的每个顶点 u , $D[u] = d_{n-1}(v, u) = d(v, u)$ 。 ■

因此, Bellman-Ford算法是正确的, 甚至还给出了一种确定有向图中是否包含负权值回路的方法。Bellman-Ford算法的运行时间易于分析。执行主for循环 $n-1$ 次, 对于 \vec{G} 中的每条边, 每个这样的循环所需时间为 $O(1)$ 。于是, 算法的运行时间为 $O(mn)$ 。概括如下:

定理7.3 给定具有 n 个顶点、 m 条边的加权有向图 \vec{G} , 以及 \vec{G} 中的一个顶点 v , 计算从 v 到 \vec{G} 中所有其他顶点的距离, 或确定 \vec{G} 中是否包含负权值回路的Bellman-Ford算法的运行时间为 $O(mn)$ 。

7.1.3 有向无环图中的最短路径

如上所述, Dijkstra算法和Bellman-Ford算法适合于有向图。但是, 如果有向图中不存在有向回路, 即它是加权有向无环图 (DAG), 则可能存在解单源点最短路径问题的更快算法。

回忆6.4.4节可知, DAG \vec{G} 的一个拓扑序列是其顶点的列表 (v_1, v_2, \dots, v_n) , 满足如果 (v_i, v_j) 是 \vec{G} 中的一条边, 那么 $i < j$ 。同时, 回忆可知, 可以利用深度优先查找算法, 计算具有 n 个顶点、 m 条边的DAG \vec{G} 中的一个拓扑序列, 其运行时间为 $O(n+m)$ 。有趣的是, 给出这样一个加权的DAG \vec{G} 的拓扑序列, 就能计算出从给定顶点 v 的所有最短路径, 所需时间为 $O(n+m)$ 。

计算DAG中最短路径的细节

算法7-3中给出的方法涉及按照拓扑序列访问 \vec{G} 中的顶点, 且对于每次访问都松弛出边。

算法7-3 有向无环图的最短路径算法

算法 DAGShortestPaths(\vec{G}, s):

输入: 具有 n 个顶点、 m 条边的加权有向无环图 (DAG) \vec{G} , 以及 \vec{G} 中的一个特殊

顶点 s

输出: 对于 \vec{G} 中的每个顶点 u , 输出标记 $D[u]$, 满足 $D[u]$ 是 \vec{G} 中从 v 到 u 的距离

计算 \vec{G} 的一个拓扑序列 (v_1, v_2, \dots, v_n)

$D[s] \leftarrow 0$

for \vec{G} 中的每个顶点 $u \neq s$ do

$D[u] \leftarrow +\infty$

for $i \leftarrow 1$ to $n-1$ do

{松弛 v_i 的每条出边}

for v_i 的每条出边 (v_i, u) do

if $D[v_i] + w((v_i, u)) < D[u]$ then

$D[u] \leftarrow D[v_i] + w((v_i, u))$

输出距离标记 D 作为距 s 的距离

有向无环图的最短路径算法的运行时间易于分析。假设利用邻接表表示有向图, 处理每个顶点的时间为常数时间再加上与其出边数成正比的时间。此外, 观察可见, 计算 \vec{G} 中顶点的拓扑序列, 其运行时间为 $O(n+m)$ 。因此, 整个算法的运行时间为 $O(n+m)$ 。图7-7说明了这个算法的运行过程。

352

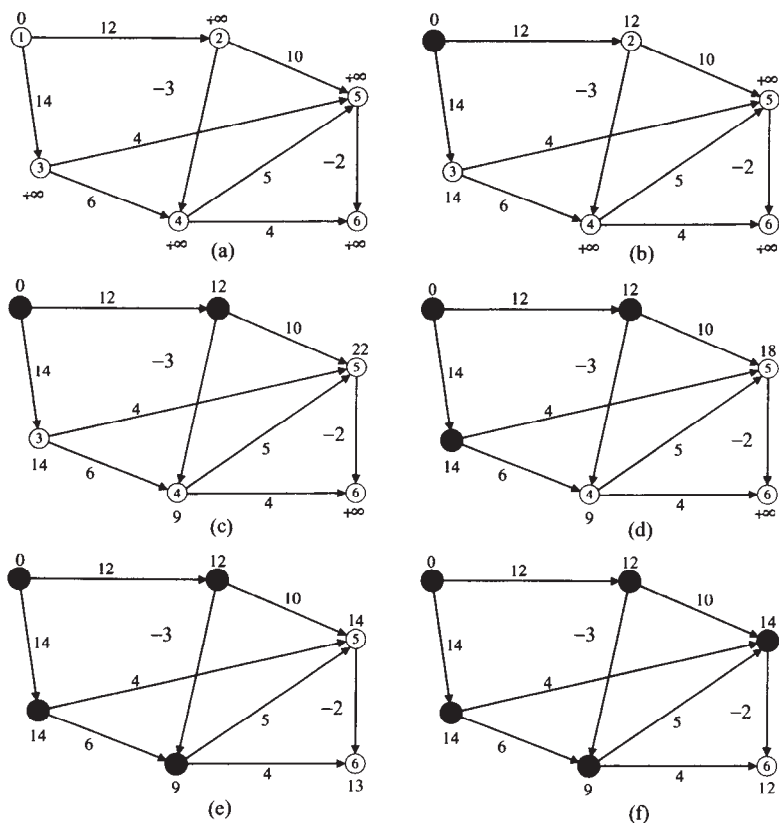


图7-7 DAG的最短路径算法的说明

定理7.4 DAGShortestPaths计算从起始顶点 s 到具有 n 个顶点、 m 条边的有向图 \vec{G} 中的每个其他顶点的距离, 所需时间为 $O(n+m)$ 。

证明 用反证法。假定 v_i 是拓扑序列中的第一个顶点, 满足 $D[v_i]$ 不是从 s 到 v_i 的距离。首先, 注意 $D[v_i] < +\infty$, 因为除 s 之外每个顶点的初始 D 值为 $+\infty$, 如果发现一条出自 s 的路径, 标记 D 的值只会更小。因此, 如果 $D[v_i] = +\infty$, 那么 v_i 由 s 不可达。于是, v_i 由 s 可达, 因而存在从 s 到 v_i 的最短路径。设 v_k 是从 s 到 v_i 的最短路径上的倒数第二个顶点。因为顶点按照拓扑次序编号, 则有 $k < i$ 。因此, $D[v_k]$ 是正确的 (可能有 $v_k = s$)。但当处理 v_k 时, 我们松弛 v_k 的每条出边, 包括从 v_k 到 v_i 的最短路径上的边。因此, $D[v_i]$ 被赋予从 s 到 v_i 的距离。但这与 v_i 的定义相矛盾; 因此, 这样的顶点 v_i 不存在。 ■

353

7.2 所有顶点对之间的最短路径

假定希望计算具有 n 个顶点、 m 条边的有向图 \vec{G} 中每一对顶点之间的最短路径的距离。当然, 如果 \vec{G} 中不含具有负权值的边, 那么对于 \vec{G} 中的每个顶点, 依次运行Dijkstra算法即可。假设用邻接表结构表示图 \vec{G} , 则这种方法的运行时间为 $O(n(n+m)\log n)$ 。在最坏情况下, 这个界限可大至 $O(n^3 \log n)$ 。同样, 如果 \vec{G} 不含负权值回路, 那么, 对于 \vec{G} 中的每个顶点, 也可依次运行Bellman-Ford算法。这种方法的运行时间为 $O(n^2m)$, 在最坏情况下为 $O(n^4)$ 。在这一节里, 考虑解所有顶点对之间最短路径问题的算法, 即使有向图包含负权值边 (但不含负权值回路), 该算法的运行时间也为 $O(n^3)$ 。

7.2.1 动态规划最短路径算法

本节讨论的第一个所有顶点对之间最短路径的算法是本书早先研究的一个算法 (即计算有向图传递闭包的Floyd-Warshall算法 (算法6-4)) 的变体。

设 \vec{G} 是给定的加权有向图。将 \vec{G} 中的顶点任意编号为 (v_1, v_2, \dots, v_n) 。正像任何动态规划算法 (5.3节) 一样, 算法的关键构造在于定义一个参数化的成本函数, 它易于计算且允许最终计算出问题的解。在这种情况下, 使用成本函数 $D_{i,j}^k$, 它定义为从 v_i 到 v_j 的距离, 中间经过的顶点在集合 $\{v_1, v_2, \dots, v_k\}$ 中。初始时,

$$D_{i,j}^0 = \begin{cases} 0 & \text{如果 } i = j \\ w((v_i, v_j)) & \text{如果 } (v_i, v_j) \text{ 是 } \vec{G} \text{ 中的一条边} \\ +\infty & \text{其他情况} \end{cases}$$

给出这个参数化的成本函数 $D_{i,j}^k$ 及其初始值 $D_{i,j}^0$, 然后对于任意 $k > 0$, 可以定义 $D_{i,j}^k$ 如下

$$D_{i,j}^k = \min \{ D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1} \}$$

换句话说, 利用编号为 $1 \sim k$ 的顶点, 从 v_i 到 v_j 的成本函数等于两条可能路径中的较小者。第一条路径只是从 v_i 到 v_j 的最短路径, 利用编号为 $1 \sim k-1$ 的顶点。第二条路径是从 v_i 到 v_k 的最短路径与从 v_k 到 v_j 的最短路径之和, 这两条最短路径都利用编号为 $1 \sim k-1$ 的顶点。此外, 不存在使用编号为 $\{v_1, v_2, \dots, v_k\}$ 的顶点的从 v_i 到 v_j 的比这两条路径更短的路径。如果存在这样一条更短的路径, 并且它不包括 v_k , 那么这就违反了 $D_{i,j}^{k-1}$ 的定义。如果存在这样一条更短的路径并且它包括 v_k , 那么它也将违反 $D_{i,k}^{k-1}$ 或 $D_{k,j}^{k-1}$ 的定义。事实上, 注意即使 \vec{G} 中存在具有负成本的边, 这个论证过程仍然成立, 只要不存在具有负成本的回路即可。在算法7-4中, 表明了如何利用这个成本函数定义构造一个所有顶点对之间最短路径问题的有效算法。

354

算法7-4 在一个不含负回路的有向图中, 计算所有顶点对之间最短路径距离的动态规划算法

算法 AllPairsShortestPaths(\vec{G}):

输入: 简单加权有向图 \vec{G} , 且图中不含负权值的回路

```

输出:  $\vec{G}$  中顶点的编号  $v_1, v_2, \dots, v_n$  和一个矩阵  $D$ , 满足  $D[i, j]$  是  $\vec{G}$  中从  $v_i$  到  $v_j$  的距离
设将  $\vec{G}$  中顶点任意编号为  $v_1, v_2, \dots, v_n$ 
for  $i \leftarrow 1$  to  $n$  do
  for  $j \leftarrow 1$  to  $n$  do
    if  $i = j$  then
       $D^0[i, i] \leftarrow 0$ 
    if  $(v_i, v_j)$  是  $\vec{G}$  中的一条边 then
       $D^0[i, j] \leftarrow w((v_i, v_j))$ 
    else
       $D^0[i, j] \leftarrow +\infty$ 
  for  $k \leftarrow 1$  to  $n$  do
    for  $i \leftarrow 1$  to  $n$  do
      for  $j \leftarrow 1$  to  $n$  do
         $D^k[i, j] \leftarrow \min\{D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j]\}$ 
  return 矩阵  $D^n$ 

```

这个动态规划算法的运行时间显然为 $O(n^3)$ 。因此, 可得以下定理:

定理 7.5 给定具有 n 个顶点的加权有向图 \vec{G} , 且图中不含负权值回路。算法 7-4 (AllPairsShortest Paths) 计算 \vec{G} 中每一对顶点之间的最短路径距离, 其运行时间为 $O(n^3)$ 。

7.2.2 利用矩阵相乘计算最短路径

可把计算有向图 \vec{G} 中所有顶点对之间的最短路径距离问题看作矩阵问题。在这一小节里, 描述一个如何利用这种方法来求解所有顶点对之间最短路径问题的 $O(n^3)$ 时间的算法。首先描述如何利用这种方法解所有顶点对之间的最短路径问题, 其运行时间为 $O(n^4)$, 然后进一步深入研究这个问题, 改进这个算法, 使其运行时间达到 $O(n^3)$ 。把矩阵相乘方法应用于最短路径问题对于在用邻接矩阵数据结构表示图的环境中特别有用。

355

1. 加权邻接矩阵表示

把 \vec{G} 中的顶点编号为 $(v_0, v_1, v_2, \dots, v_{n-1})$, 按照约定顶点从下标 0 开始编号。给定 \vec{G} 中顶点的这种编号方式, 存在图的一种自然的加权邻接矩阵表示法, 其中定义 $A[i, j]$ 如下:

$$A[i, j] = \begin{cases} 0 & \text{如果 } i = j \\ w((v_i, v_j)) & \text{如果 } (v_i, v_j) \text{ 是 } \vec{G} \text{ 中的一条边} \\ +\infty & \text{其他情况} \end{cases}$$

(注意这个定义与上一节定义的成本函数 $D_{i,j}^0$ 相同)。

2. 最短路径和矩阵相乘

换句话说, $A[i, j]$ 存储从 v_i 到 v_j 的最短路径距离, 它使用了这条路径中的一条或少数几条边。于是, 利用矩阵 A 定义另一个矩阵 A^2 , 满足 $A^2[i, j]$ 存储从 v_i 到 v_j 的最短路径距离, 它使用了这条路径中的至多两条边。至多有两条边的路径要么为空 (0 条边的路径), 要么向 0 条边的路径或 1 条边的路径添加一条边。于是, 定义 $A^2[i, j]$ 如下

$$A^2[i, j] = \min_{l=0,1,\dots,n-1} \{A[i, l] + A[l, j]\}$$

因此, 给定 A , 利用类似于标准矩阵相乘算法的算法计算矩阵 A^2 的时间为 $O(n^3)$ 。

事实上, 可把这个计算看作一个矩阵相乘, 只要简单地重新定义矩阵相乘算法中的 “+” 和

“ \times ”运算符的含义即可（程序设计语言C++中允许重载这样的运算符）。如果把“+”重新定义为“min”，把“ \times ”重新定义为“+”，那么可把 $A^2[i, j]$ 重写为一个真正的矩阵相乘：

$$A^2[i, j] = \sum_{l=0, 1, \dots, n-1} A[i, l] \cdot A[l, j]$$

实际上，从矩阵相乘的观点可以看出我们为什么把这个矩阵写为“ A^2 ”的原因，因为它是矩阵 A 的平方。

继续用这个方法定义矩阵 A^k ，满足 $A^k[i, j]$ 是存储从 v_i 到 v_j 的最短路径距离，在这条路径中至多利用 k 条边。因为至多有 k 条边的路径等价于至多有 $k-1$ 条边的路径再加上可能一条额外的边，可以定义 A^k 如下

$$A^k[i, j] = \sum_{l=0, 1, \dots, n-1} A^{k-1}[i, l] \cdot A[l, j]$$

356

继续运算符的重新定义，使“+”代表“min”，“ \cdot ”代表“+”。

如果 \vec{G} 不含负权值回路，那么 A^{n-1} 存储 \vec{G} 中每对顶点之间的最短路径的距离。这个观察依据以下事实，即任何良好定义的最短路径至多包含 $n-1$ 条边。如果一条路径中的边数超过 $n-1$ 条，则某些顶点一定会发生重复；因此，它一定包含一个回路。但是最短路径永远不会包含回路（除非 \vec{G} 中存在负权值的回路）。因此，要求解所有顶点对之间的最短路径问题，所需做的全部工作就是让 A 自乘 $n-1$ 次。因为每个这样的乘法的运行时间为 $O(n^3)$ ，由此得以下定理：

定理7.6 给定一个具有 n 个顶点的加权有向图 \vec{G} ，图中不含具有负权值的回路，并给定 \vec{G} 的加权邻接矩阵 A ，则 \vec{G} 的所有顶点对之间的最短路径问题可通过计算 A^{n-1} 求解，其运行时间为 $O(n^4)$ 。

在10.1.4节中，讨论了数的指数算法，它可用于目前矩阵相乘的环境中，计算 A^{n-1} 所需时间为 $O(n^3 \log n)$ 。但是，在所有顶点对的最短路径问题中，通过利用一些额外的结构，计算 A^{n-1} 所需时间可变为 $O(n^3)$ 。

3. 矩阵闭包

如上所述，如果 \vec{G} 中不含负权值回路，那么 A^{n-1} 对 \vec{G} 中每对顶点之间的最短路径距离进行编码。一条良好定义的最短路径不能包含回路；因此，受限至多包含 $n-1$ 条边的最短路径一定是一条真正最短路径。同样，至多包含 n 条边的最短路径也一定是一条真正最短路径，至多包含 $n+1$ 条边、 $n+2$ 条边等的最短路径也是如此。于是，如果 \vec{G} 中不含负权值回路，那么

$$A^{n-1} = A^n = A^{n+1} = A^{n+2} = \dots$$

矩阵 A 的闭包（closure）定义如下

$$A^* = \sum_{i=0}^{\infty} A^i$$

条件是这样的矩阵存在。如果 A 是一个加权邻接矩阵，那么， $A^*[i, j]$ 是从 v_i 到 v_j 的所有可能路径之和。在这种情况下， A 是有向图 \vec{G} 的加权邻接矩阵，重新定义“+”为“min”。因此，可以重写 A^* 为

$$A^* = \min_{i=0, \dots, \infty} \{A^i\}$$

此外，由于要计算最短路径距离， A^{i+1} 中的元素永远不会大于 A^i 中的元素。于是，对于不含负权值的回路且具有 n 个顶点的有向图 \vec{G} ，它的加权邻接矩阵为

$$A^* = A^{n-1} = A^n = A^{n+1} = A^{n+2} = \dots$$

357

即 $A^*[i, j]$ 存储从 v_i 到 v_j 的最短路径的长度。

4. 计算加权邻接矩阵的闭包

可用分治法在 $O(n^3)$ 时间内计算闭包 A^* 。不失一般性, 可以假设 n 是2的幂(否则, 在有向图 \vec{G} 中补上没有入边或出边的额外顶点)。把 \vec{G} 中顶点的集合 V 分成两个大小相等的集合 $V_1 = \{v_0, \dots, v_{n/2-1}\}$ 和 $V_2 = \{v_{n/2}, \dots, v_{n-1}\}$ 。给定这个划分, 同样可以把邻接矩阵 A 分成四块 B 、 C 、 D 和 E , 每块大小为 $n/2$ 行和 $n/2$ 列, 定义如下:

- B : 从 V_1 到 V_1 的边的权值。
- C : 从 V_1 到 V_2 的边的权值。
- D : 从 V_2 到 V_1 的边的权值。
- E : 从 V_2 到 V_2 的边的权值。

即

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$$

图7-8说明了这四个边的集合。

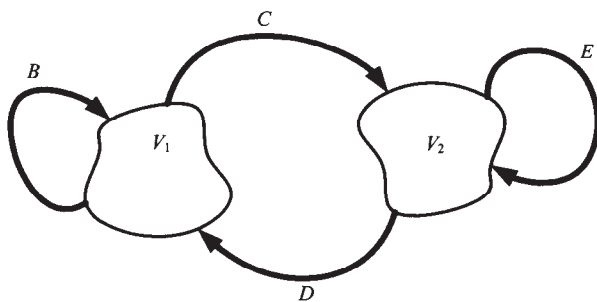


图7-8 在基于分治法计算 A^* 的过程中, 利用边的4个集合划分邻接矩阵 A 的说明

同样, 可以把 A^* 分成四块 W 、 X 、 Y 和 Z , 其定义类似, 如下:

- W : 从 V_1 到 V_1 的最短路径的权值。
- X : 从 V_1 到 V_2 的最短路径的权值。
- Y : 从 V_2 到 V_1 的最短路径的权值。
- Z : 从 V_2 到 V_2 的最短路径的权值。

即

$$A^* = \begin{pmatrix} W & X \\ Y & Z \end{pmatrix}$$

358

5. 子矩阵方程

由上述定义, 可以从子矩阵 B 、 C 、 D 和 E 直接导出定义 W 、 X 、 Y 和 Z 的简单方程。

- 对于 W 中由子路径闭包组成的路径, $W = (B + C \cdot E^* \cdot D)^*$, 该路径要么停留在 V_1 中要么跳到 V_2 上, 并在 V_2 上漫游一段时间, 然后再跳回到 V_1 上。
- 对于 X 中由子路径闭包组成的路径, $X = W \cdot C \cdot E^*$, 这些子路径在 V_1 中(可能跳到 V_2 又返回)开始和结束, 紧接着跳到 V_2 和停留在 V_2 中的子路径闭包上。
- 对于 Y 中由子路径闭包组成的路径, $Y = E^* \cdot D \cdot W$, 这些子路径停留在 V_2 中, 紧接着跳到 V_1

和在 V_1 中开始和结束的子路径闭包上。

- 对于 Z 中由路径组成的路径, $Z = E^* + E^* \cdot D \cdot W \cdot C \cdot E^*$, 这些路径停留在 V_2 中或漫游在 V_2 的路径上, 它会跳到 V_1 上, 并在 V_1 上漫游一段时间 (可能跳到 V_2 又返回), 跳回到 V_2 上, 然后停留在 V_2 中。

给出这些方程, 容易构造计算 A^* 的一个递归算法。在这个算法中, 把 A 划分成上述的块 B 、 C 、 D 和 E 。然后递归计算闭包 E^* 。给定 E^* , 就能递归计算闭包 $(B + C \cdot E^* \cdot D)^*$, 它就是 W 。

注意不需递归计算其他闭包, 即可计算 X 、 Y 和 Z 。因此, 在执行常数个矩阵加法和乘法之后, 就可以计算出 A^* 中所有的块。由此得到以下定理。

定理7.7 给定具有 n 个顶点的加权有向图 \vec{G} , 图中不含负权值回路, 并给定 \vec{G} 的加权邻接矩阵 A , 则 \vec{G} 的所有顶点对之间的最短路径问题可通过计算 A^* 求解, 其运行时间为 $O(n^3)$ 。

证明 上面已经论证为什么计算 A^* 能够解决所有顶点对之间的最短路径问题。然后考虑计算 A^* 的分治法的运行时间, 即计算 $n \times n$ 邻接矩阵 A 的闭包的分治法的运行时间。这个算法由两次递归调用计算 $n/2 \times n/2$ 子矩阵, 以及常数个矩阵加法和乘法组成 (利用“min”代表“+”, “+”代表“·”)。因此, 假设利用 $O(n^3)$ 时间的直接矩阵相乘算法, 那么可以表征计算 A^* 的运行时间 $T(n)$ 为

$$T(n) = \begin{cases} b & \text{如果 } n=1 \\ 2T(n/2) + cn^3 & \text{如果 } n>1 \end{cases}$$

[359] 其中 $b > 0$ 且 $c > 0$ 是常数。于是, 由主定理(5.3)可知, 计算 A^* 时间为 $O(n^3)$ 。 ■

7.3 最小生成树

假定希望利用最少量的电缆线, 连接一座新建筑物中的所有计算机。同样, 假定有一个无向计算机网络, 在这个网络中, 两个路由器之间的每条连接都有一个使用成本; 希望以可能最少的成本连接所有路由器。可以利用一个加权图 G 为这些问题建模, 在这个加权图中, 顶点表示计算机或者路由器, 边表示所有可能的计算机对 (u, v) , 其中边 (v, u) 上的权值 $w((v, u))$ 等于连接计算机 v 与 u 所需的电缆量或者网络成本。我们不是计算从某个特殊顶点 v 开始的一棵最短路径树, 而是找出一棵 (自由) 树 T , 它包含 G 中的所有顶点, 且使这棵树上的总权值达到最小。本节集中考察找出这样一棵树的方法。

1. 问题定义

给出一个加权无向图 G , 希望找到这样一棵树 T , 它包含 G 中所有顶点, 且使 T 中各条边的权值之和达到最小, 即

$$w(T) = \sum_{e \in T} w(e)$$

由6.1节可知, 像这样的一棵树 (它包含一个连通图 G 中的每个顶点) 称为生成树 (spanning tree)。计算具有最小总权值的一棵生成树 T 是要构造一棵最小生成树 (或MST) 的问题。

研究最小生成树问题的有效算法, 可以提前计算机科学自身的现代概念, 在这一节里, 讨论解MST问题的两个算法。这些算法都是贪心法的经典应用。正如5.1节中讨论的那样。利用贪心法, 反复选择对象加入一个增长的集合中, 选择一个对象的方式是: 使得某个成本函数最小化。

讨论的第一个MST算法是Kruskal算法, 它按照边的权值大小考虑每一条边, 并按簇“增长”

MST。讨论的第二个MST算法是Prim-Jarník算法，从单独一个根顶点开始，“增长”MST集，这与Dijkstra最短路径算法的执行方式非常相似。通过讨论第三个方法结束本节，该算法由Barůvka提出，它以并行方式应用贪心法。

如7.1.1节所述，为了简化算法的描述，以下假设输入图 G 是无向的（即图中所有边都是无向的），并且是简单图（即图中不含自环，也不含平行边）。因此，表示 G 中的边为无序顶点对 (u, z) 。

360

2. 关于最小生成树的关键事实

但是，在讨论这些算法的细节之前，给出关于最小生成树的一个关键事实，它构成最小生成树算法的基础。特别是，这里讨论的所有MST算法都基于贪心法，在这种情况下，它主要依赖于以下事实（如图7-9所示）。

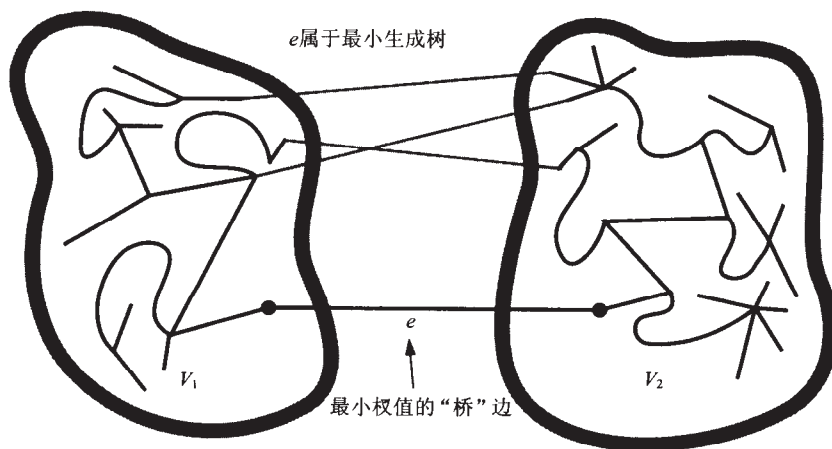


图7-9 关于最小生成树的关键事实的说明

定理7.8 设 G 是一个加权连通图， V_1 和 V_2 是 G 的顶点的一个划分，它把 G 的顶点分成两个不相交的非空子集。进一步设 e 是 G 中具有最小权值的边，该边的一个端点在 V_1 中，另一个端点在 V_2 中。存在一棵最小生成树，包含 e 作为它的一条边。

证明 设 T 是 G 的一棵最小生成树。如果 T 中没有包含边 e ，那么向 T 中添加 e 必定会构成一个回路。于是，这个回路中存在某条边 f ，它的一个端点在 V_1 中，另一个端点在 V_2 中。此外，根据 e 的选择， $w(e) \leq w(f)$ 。如果从 $T \cup \{e\}$ 中删除 f ，则得到一棵生成树，它的总权值不超过以前的总权值。因为 T 是一棵最小生成树，这棵新树也必定为一棵最小生成树。 ■

事实上，如果 G 中的权值互不相同，那么最小生成树是唯一的；这个关键事实的证明留作习题（C-7.5）。

此外，注意即使图 G 包含负权值边或负权值回路，定理7.8仍然成立。这与提出的最短路径算法不同。

361

7.3.1 Kruskal 算法

定理7.8如此重要的原因在于，它可用作构建一棵最小生成树的基础。在Kruskal算法中，它用于按照簇构建MST树。起初，每个顶点自身就是一个簇。然后，算法按照权值的增序依次考虑每条边。如果一条边 e 连接两个不同的簇，那么将 e 添加到最小生成树的边集合中，并且通过 e 将两个簇合并为一个簇。另一方面，如果 e 所连接的顶点已在同一个簇中，那么将 e 丢弃。一旦算法

添加的边足以形成一棵生成树，则算法终止，并输出这棵树作为最小生成树。

算法7-5给出了求解MST问题的Kruskal算法的伪代码。图7-10、图7-11和图7-12显示了该算法的工作过程。

算法7-5 MST问题的Kruskal算法

算法 KruskalMST(G):
输入: 给定具有 n 个顶点、 m 条边的简单连通加权图 G
输出: G 的最小生成树 T
for G 中的每个顶点 v **do**
 定义基本簇 $C(v) \leftarrow \{v\}$.
初始化优先队列 Q ，包含 G 中的所有边，并利用权值作为关键字
 $T \leftarrow \emptyset$ { T 最终将包含MST中的边}
while T 中的边数少于 $n-1$ **do**
 $(u, v) \leftarrow Q.\text{removeMin}()$
 令 $C(v)$ 为包含 v 的簇， $C(u)$ 为包含 u 的簇
 if $C(v) \neq C(u)$ **then**
 向 T 中添加一条边 (v, u)
 把 $C(v)$ 和 $C(u)$ 合并为一个簇，即进行 $C(v)$ 和 $C(u)$ 的并操作
return 树 T

362

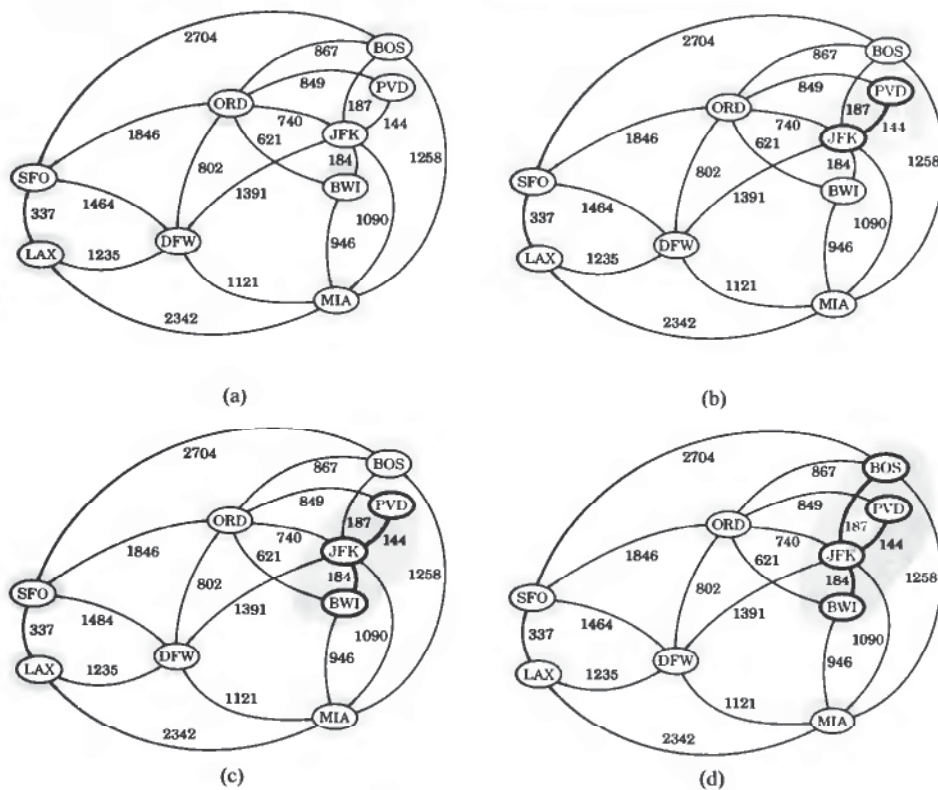


图7-10 Kruskal MST算法在具有整型权值的图上的执行示例。每个簇用阴影区域表示，每次迭代中考虑的边用深色线条突出显示（图7-11继续）

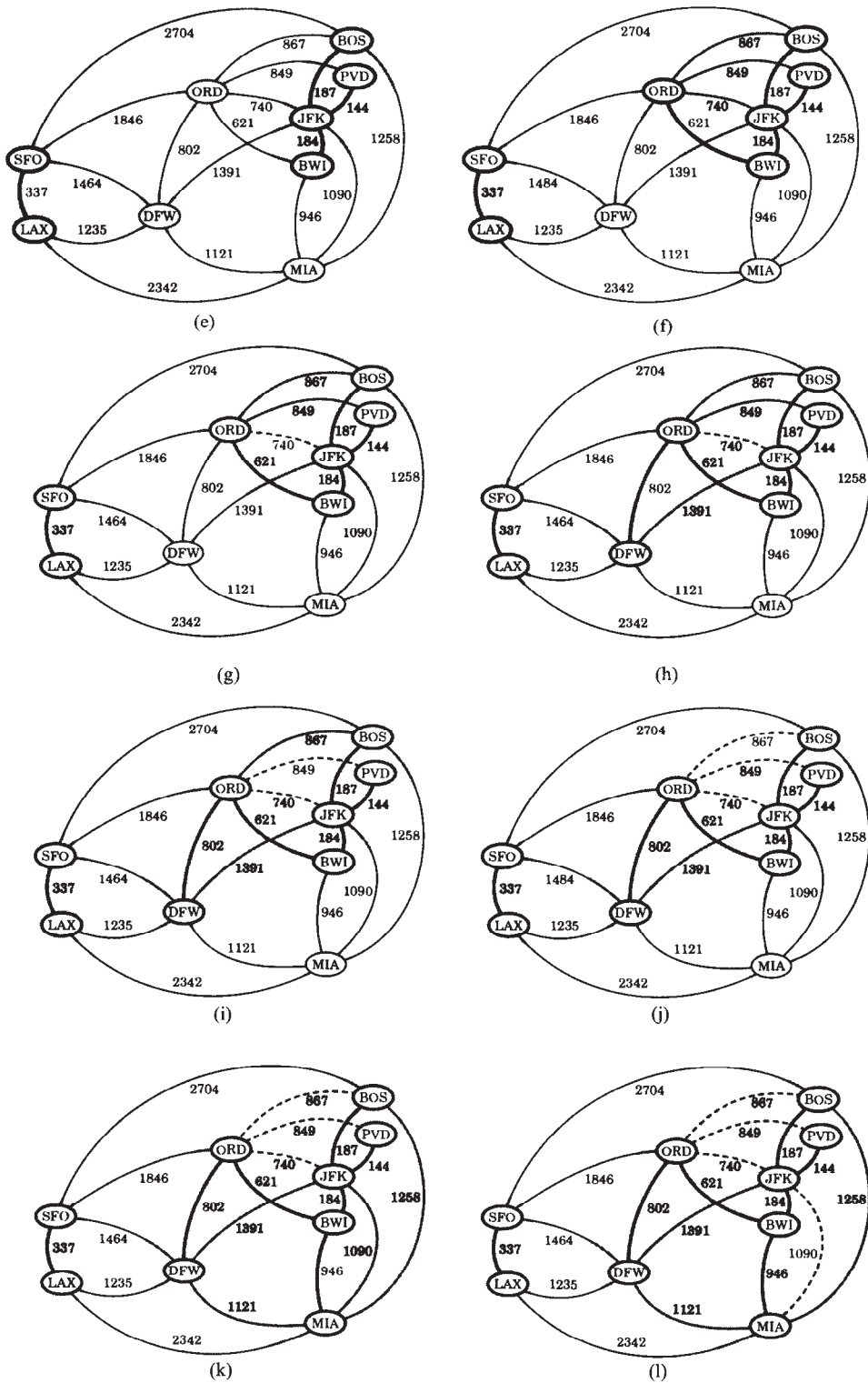


图7-11 Kruskal MST算法的执行示例（续），虚线表示被拒绝的边（图7-12继续）

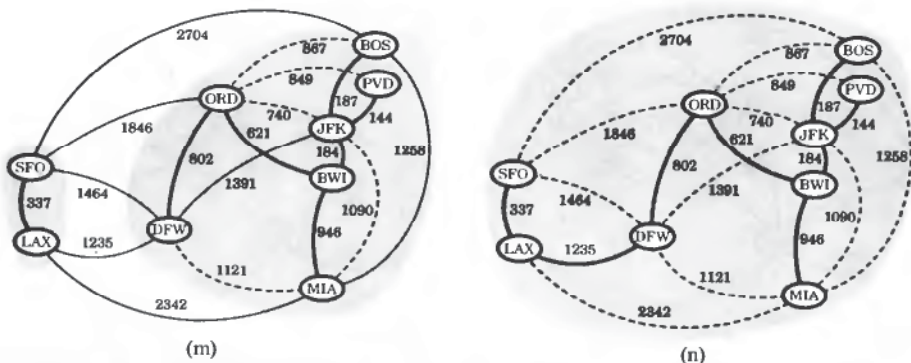


图7-12 Kruskal MST算法的执行示例 (图7-10和图7-11续), (n)中考虑的边把最后两个簇合并起来, 结束Kruskal算法的执行过程

如前所述, Kruskal算法的正确性可由关于最小生成树的关键事实 (即定理7.8) 导出。每当Kruskal算法向最小生成树 T 中添加一条边 (v, u) 时, 可以定义顶点 V 的集合的一个划分 (如在定理中所述的那样), 设 V_1 为包含 v 的簇, V_2 为包含 u 中其余顶点的簇。这显然定义了 V 中顶点的一个不相交划分。更重要的是, 因为按照权值次序从 Q 中提取边, 因此 e 必定是一条具有最小权值的边, 它的一个顶点在 V_1 中, 另一个顶点在 V_2 中。因此, Kruskal算法总是会添加一条有效的最小生成树的边。

1. Kruskal算法的实现

设 n 和 m 分别表示输入图 G 中的顶点数和边数。假设可以用常数时间比较边的权值。由于算法7-5给出的是Kruskal算法的高级描述, 因此分析它的运行时间需要给出实现的细节。确切地讲, 应该说明使用的数据结构及其实现方式。

利用堆实现优先队列 Q 。因此, 通过反复插入过程, 可在 $O(m \log m)$ 时间内初始化 Q , 或者利用自底向上构造堆的过程 (2.4.4节), 可在 $O(m)$ 时间内初始化 Q 。此外, 在while循环的每次迭代中, 删除一条具有最小权值的边的时间为 $O(\log m)$, 由于 G 是简单图, 它实际上为 $O(\log n)$ 。

2. 一种简单的簇合并策略

对于簇利用基于表的划分实现 (4.2.2节), 即用一个顶点组成的无序链表表示每个簇 C , 该链表利用每个顶点 v 存储指向其簇 $C(v)$ 的引用。有了这种表示法, 测试是否 $C(v) \neq C(u)$ 所需的时间为 $O(1)$ 。当需合并两个簇 $C(v)$ 与 $C(u)$ 时, 把较小簇中的元素移到较大的簇中, 并更新较小簇中顶点的簇引用。由于可以简单地把较小簇中的元素添加到较大簇中的表尾, 因而, 合并两个簇所需的时间与较小簇的大小成正比, 即合并两个簇 $C(v)$ 和 $C(u)$ 所需时间为 $O(\min\{|C(u)|, |C(v)|\})$ 。还有其他一些更有效的方法用于合并两个簇 (见4.2.2节), 但这里给出的简单方法就足够了。

引理7.3 考虑在 n 个顶点的图上执行Kruskal算法, 其中用序列及每个顶点的簇引用表示簇。合并簇所需的总时间为 $O(n \log n)$ 。

证明 观察每次将一个顶点移到一个新簇中的情况。包含该顶点的簇的大小至少加倍。设 $t(v)$ 表示将顶点 v 移到新簇中的次数, 因为最大簇的大小为 n 。则有

$$t(v) \leq \log n$$

在Kruskal算法中, 合并簇所花费的总时间为在每个顶点上所做的工作之和, 它与以下数量成正比:

$$\sum_{v \in G} t(v) \leq n \log n$$

利用引理7.3以及 Dijkstra算法分析中使用的类似论证,可知Kruskal算法的运行时间为 $O((n+m) \log n)$, 由于 G 是简单、连通图, 这可简化为 $O(m \log n)$ 。

定理7.9 给定具有 n 个顶点、 m 条边的简单连通加权图 G , Kruskal算法构造 G 的一棵最小生成树所需的时间为 $O(m \log n)$ 。

7.3.2 Prim-Jarník 算法

在Prim-Jarník算法中, 从某个“根”顶点为 v 的单一簇开始, 生长出一棵最小生成树。其主要思想类似于Dijkstra算法。从某个顶点 v 开始, 定义顶点的初始云集 C 。然后, 在每次迭代中, 选择一条具有最小权值的边 $e = (v, u)$, 连接云集 C 中的一个顶点 v 与 C 外的一个顶点 u 。这样, 顶点 u 被带入云集 C 中, 重复这个过程, 直到形成生成树。再一次, 关于最小生成树的关键事实起作用, 因为通过总是选择一条具有最小权值的边, 连接 C 内的一个顶点与 C 外的一个顶点, 确信总会向MST中添加一条有效边。

1. 生长出单一MST

为了有效地实现这种方法, 可以从Dijkstra算法中得到另一个提示。维持云集 C 外每个顶点 u 的标记 $D[u]$, 使得 $D[u]$ 存储连接 u 与云集 C 的当前最佳边的权值。在决定哪一个顶点是下一个要加入云集中的顶点时, 这些标记可使我们减少必须考虑的边数。算法7-6给出这种方法的伪代码。

366

算法7-6 MST问题的Prim-Jarník算法

算法 PrimJarníkMST(G):

输入: 具有 n 个顶点、 m 条边的加权连通图 G

输出: G 的最小生成树 T

选取 G 中的任意顶点 v

$D[v] \leftarrow 0$

for 每个顶点 $u \neq v$ **do**

$D[u] \leftarrow +\infty$

初始化 $T \leftarrow \emptyset$

对于每个顶点 u , 用数据项 $((u, \text{null}), D[u])$ 初始化优先队列 Q , 其中 (u, null) 是元素, $D[u]$ 是关键字

while Q 非空 **do**

$(u, e) \leftarrow Q.\text{removeMin}()$

 向 T 中添加顶点 u 和边 e

for u 的每个邻接顶点 z , 且 z 在 Q 中 **do**

 {对边 (u, z) 进行松弛过程}

if $w((u, z)) < D[z]$ **then**

$D[z] \leftarrow w((u, z))$

 将 Q 中顶点 z 的元素更改为 $(z, (u, z))$

 将 Q 中顶点 z 的关键字更改为 $D[z]$

return 树 T

2. 分析Prim-Jarník算法

设 n 和 m 分别表示输入图 G 中的顶点数和边数。Prim-Jarník算法的实现问题类似于Dijkstra算法的那些实现问题。如果利用堆实现优先队列 Q , 且堆支持基于定位器的优先队列方法(见2.4.4节), 则在每次迭代中, 就能在 $O(\log n)$ 时间内提取顶点 u 。

此外, 可在 $O(\log n)$ 时间内更新每个 $D[z]$ 值, 同时, 对于每条边 (u, z) , 这是一个至多考虑一次

的计算。每次迭代中的其他步骤可在常数时间内实现。因此，总运行时间为 $O((n+m)\log n)$ ，即为 $O(m\log n)$ 。因此，概括如下：

定理7.10 给定具有 n 个顶点、 m 条边的简单连通加权图 G ，Prim-Jarnik算法构造 G 的一棵最小生成树所需的时间为 $O(m\log n)$ 。

图7-13和图7-14说明了Prim-Jarnik算法的执行过程。

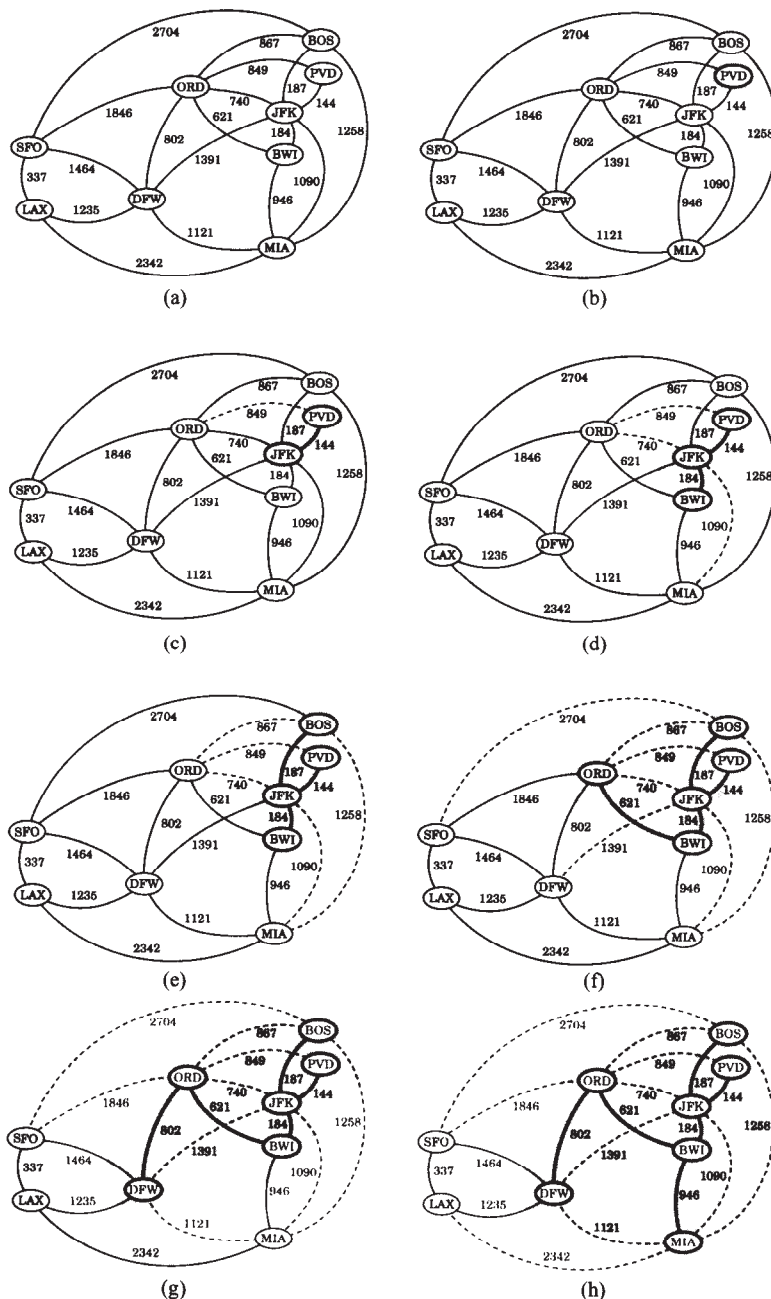


图7-13 Prim-Jarnik算法的可视化（图7-14续）

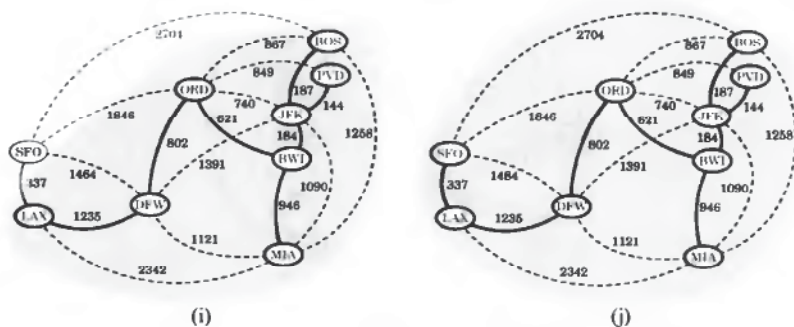


图7-14 Prim-Jarník算法的可视化（接图7-13）

7.3.3 Barůvka 算法

上面描述的每个最小生成树算法都利用优先队列 Q 得到其有效运行时间，优先队列 Q 可以利用堆（或者甚至更复杂的数据结构）实现。这种用法似乎是自然的，因为最小生成树算法涉及应用贪心法——而在这种情况下，贪心法显式地优化了所讨论的图中顶点之间的优先级。这可能有点令人惊讶，但在这一节里，实际上可以不使用优先队列来设计一个有效的最小生成树算法。此外，可能更令人惊讶的是，这个简单的方法起源于最古老的最小生成树算法——Barůvka算法。

算法7-7给出了Barůvka最小生成树算法的伪代码描述，图7-15中说明了这个算法的执行过程。

算法7-7 Barůvka算法的伪代码

算法 BarůvkaMST(G):

输入：具有 n 个顶点、 m 条边的连通加权图 $G = (V, E)$

输出： G 的最小生成树 T

设 T 是 G 的子图，最初只含 V 中的顶点

while T 中边数少于 $n-1$ { T 还不是MST} **do**

for T 中的每个连通分量 C_i **do**

 {对于簇 C_i ，执行MST边增加的过程}

 找出 E 中最小权值边 $e = (v, u)$ ，其中 $v \in C_i$ 且 $u \notin C_i$

 把 e 添加到 T 中（除非 e 已在 T 中）

return T

1. 实现Barůvka算法

Barůvka算法的实现相当简单，只需要能够做以下事情：

- 维持进行边插入的森林 T ，利用 T 的邻接表结构，轻松支持每次插入所需的时间为 $O(1)$ 。
- 遍历森林 T ，找出连通分量（簇）。利用 T 的深度优先查找，这个过程可在 $O(n)$ 时间内轻松完成。
- 利用顶点所属的簇名标记顶点，为每个顶点使用一个额外的实例变量即可。
- 找出 E 中依附于簇 C_i 的最小权值边，这可以通过为 C_i 中的顶点扫描 G 中的邻接表完成。

像Kruskal算法一样，Barůvka算法在一系列轮中增长顶点的许多簇，而不只是针对一个簇，来构建最小生成树，就像在Prim-Jarník算法中所做的那样。但在Barůvka算法中，把最小生成树的关键事实同时应用于每个簇中，这种方法使得每一轮可以增加更多的边。

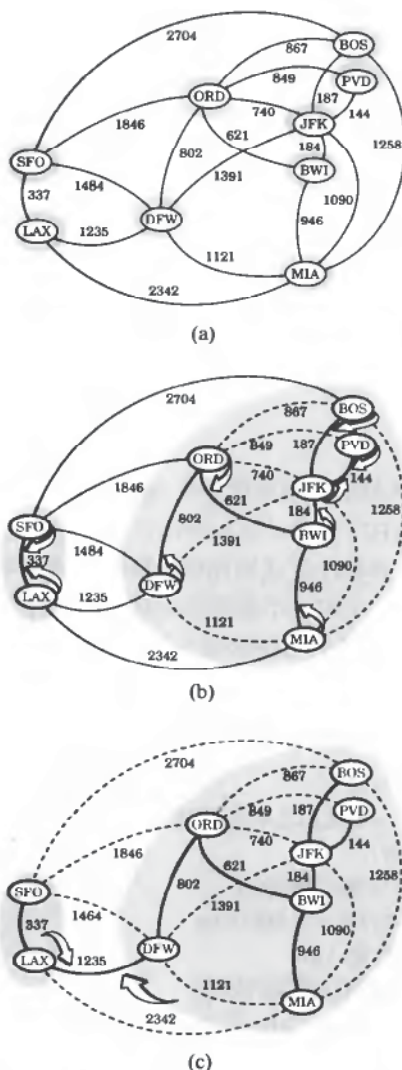


图7-15 Baruvka算法的执行示例。簇用阴影区域表示。用箭头强调每个簇选择的边，并用一条粗线画出每个这样的MST边。那些确定不在MST中的边用虚线表示

2. 为什么算法是正确的

在Baruvka算法的每次迭代中，从最小生成树边的当前集合 T 的每个连通分量中，选择最小权值的出边。在各种情况下，这条边都是有效的选择，因为如果 V 的一个划分把它的顶点分成两部分，一部分顶点在 C_i 中，另一部分在 C_i 外，那么为保证 e 属于一棵最小生成树，为 C_i 所选的边 e 应满足最小生成树的关键事实（定理7.8）。

370

3. 分析Baruvka算法

下面分析Baruvka算法（算法7-7）的运行时间。在每一轮实现中，可以通过对每个簇中顶点的邻接表的进行穷尽查找，找出每个簇中最小权值的出边，来执行查找过程。因此，查找具有最小权值的边所花费的总时间为 $O(m)$ ，这包括检查 G 中的每一条边两次：一次是查找 v ，另一次是查找 u （因为顶点用它们所在的簇编号标记）。主while循环中的其余计算包括对所有顶点进行重新标记，这需要 $O(n)$ 的时间，遍历 T 中的所有边所需时间为 $O(n)$ 。因此，Baruvka算法中的每一轮

所花费的时间为 $O(m)$ (因为 $n \leq 4m$)。在算法的每一轮中,选择每个簇中的一条出边,然后将 T 中的每个新连通分量合并进一个新簇中。因此, T 中原来的每个簇必定会与 T 中至少另一个原来的簇合并。也就是说,在Barůvka算法的每一轮中,簇的总数会减半。于是,总轮数为 $O(\log n)$;因此,Barůvka算法的总运行时间为 $O(m \log n)$ 。概括如下:

定理7.11 给定具有 n 个顶点、 m 条边的连通加权图 G , Barůvka算法计算 G 的一棵最小生成树所需的时间为 $O(m \log n)$ 。

7.3.4 MST 算法比较

尽管上述每个MST算法利用不同的数据结构和不同的方法构建最小生成树,但它们最坏情况下的运行时间相同。

考虑辅助数据结构,Kruskal算法利用优先队列存储边,用表实现的集合存储簇。Prim-Jarník算法只利用优先队列存储顶点-边对。因此,从程序设计的简易性的观点来看,Prim-Jarník算法是首选算法。实际上,Prim-Jarník算法与Dijkstra算法如此相似,以至于不费力气就可以把Dijkstra算法的实现转换为Prim-Jarník算法的实现。Barůvka算法需要表示连通分量的方式。因此,从程序设计的简易性的观点来看,Prim-Jarník算法和Barůvka算法似乎是最好的。

就常数因子而言,这三个算法相似之处在于,在它们各自的运行时间中都隐含着相对较小的常数因子。如果按照权值对边进行排序(利用4.2.2节的划分数据结构),那么Kruskal算法的隐含运行时间还可以改进。同时,只要对算法稍做修改(在习题C-7.12中探讨),Barůvka算法最坏情况下的运行时间也可改进到 $O(n^2)$ 。因此,这三个算法之间没有明显的优胜者,尽管Barůvka算法是三种算法中最容易实现的。

371
372

7.4 Java 示例: Dijkstra 算法

在这一节里,给出了执行Dijkstra算法(算法7-1)的Java代码,假设给定一个具有正整型权值的无向图。

用抽象类Dijkstra(代码段7-1~7-3)表示Dijkstra算法的实现,这个类声明了抽象方法weight(e),用于提取边 e 的权值。通过实现了方法weight(e)的子类对类Dijkstra进行扩展。例如,参见代码段7-4中的类MyDijkstra。

代码段7-1 实现Dijkstra算法的Dijkstra类(在代码段7-2和代码段7-3中继续)

```
/** Dijkstra's algorithm for the single-source shortest path problem
 * in an undirected graph whose edges have integer weights. Classes
 * extending this abstract class must define the weight(e) method,
 * which extracts the weight of an edge. */
public abstract class Dijkstra {
    /** Execute Dijkstra's algorithm. */
    public void execute(InspectableGraph g, Vertex source) {
        graph = g;
        dijkstraVisit(source);
    }
    /** Attribute for vertex distances. */
    protected Object DIST = new Object();
    /** Set the distance of a vertex. */
    protected void setDist(Vertex v, int d) {
```

```

    v.set(DIST, new Integer(d));
}
/** Get the distance of a vertex from the source vertex. This method
 * returns the length of a shortest path from the source to u after
 * method execute has been called. */
public int getDist(Vertex u) {
    return ((Integer) u.get(DIST)).intValue();
}
/** This abstract method must be defined by subclasses.
 * @return weight of edge e. */
protected abstract int weight(Edge e);
/** Infinity value. */
public static final int INFINITE = Integer.MAX_VALUE;
/** Input graph. */
protected InspectableGraph graph;
/** Auxiliary priority queue. */
protected PriorityQueue Q;

```

373

方法dijkstraVisit执行上述算法。使用的优先队列 Q 支持基于定位器的方法(2.4.4节)。当用方法insert在 Q 中插入一个顶点 u 时,返回 Q 中 u 的定位器。按照修饰模式,通过方法setLoc把定位器附着在 u 上,并利用方法getLoc检索 u 的定位器。在松弛过程中,方法replaceKey(ℓ , d)把顶点 z 的标记改为 d ,其中 ℓ 是 z 的定位器。

代码段7-2 Dijkstra类中的方法dijkstraVisit

```

/** The actual execution of Dijkstra's algorithm.
 * @param v source vertex. */
protected void dijkstraVisit (Vertex v) {
    // initialize the priority queue Q and store all the vertices in it
    Q = new ArrayHeap(new IntegerComparator());
    for (VertexIterator vertices = graph.vertices(); vertices.hasNext();) {
        Vertex u = vertices.nextVertex();
        int u_dist;
        if (u==v)
            u_dist = 0;
        else
            u_dist = INFINITE;
        // setDist(u, u_dist);
        Locator u_loc = Q.insert(new Integer(u_dist), u);
        setLoc(u, u_loc);
    }
    // grow the cloud, one vertex at a time
    while (!Q.isEmpty()) {
        // remove from Q and insert into cloud a vertex with minimum distance
        Locator u_loc = Q.min();
        Vertex u = getVertex(u_loc);
        int u_dist = getDist(u_loc);
        Q.remove(u_loc); // remove u from the priority queue
        setDist(u, u_dist); // the distance of u is final
        destroyLoc(u); // remove the locator associated with u
        if (u_dist == INFINITE)
            continue; // unreachable vertices are not processed
        // examine all the neighbors of u and update their distances
        for (EdgeIterator edges = graph.incidentEdges(u); edges.hasNext();) {
            Edge e = edges.nextEdge();
            Vertex z = graph.opposite(u,e);
            if (hasLoc(z)) { // check that z is in Q, i.e., it is not in the cloud

```

```

        int e_weight = weight(e);
        Locator z_loc = getLoc(z);
        int z_dist = getDist(z_loc);
        if ( u_dist + e_weight < z_dist ) // relaxation of edge e = (u,z)
            Q.replaceKey(z_loc, new Integer(u_dist + e_weight));
    }
}
}
}

```

374

代码段7-3 Dijkstra类中的辅助方法。假设图中的顶点是可修饰的（接代码段7-1和代码段7-2）

```

/** Attribute for vertex locators in the priority queue Q. */
protected Object LOC = new Object();
/** Check if there is a locator associated with a vertex. */
protected boolean hasLoc(Vertex v) {
    return v.has(LOC);
}
/** Get the locator in Q of a vertex. */
protected Locator getLoc(Vertex v) {
    return (Locator) v.get(LOC);
}
/** Associate with a vertex its locator in Q. */
protected void setLoc(Vertex v, Locator l) {
    v.set(LOC, l);
}
/** Remove the locator associated with a vertex. */
protected void destroyLoc(Vertex v) {
    v.destroy(LOC);
}
/** Get the vertex associated with a locator. */
protected Vertex getVertex(Locator l) {
    return (Vertex) l.element();
}
/** Get the distance of a vertex given its locator in Q. */
protected int getDist(Locator l) {
    return ((Integer) l.key()).intValue();
}
}

```

代码段7-4 MyDijkstra类扩展了Dijkstra，并提供了方法weight(e)的一个具体实现

```

/** A specialization of class Dijkstra that extracts edge weights from
 * decorations. */
public class MyDijkstra extends Dijkstra {
    /** Attribute for edge weights. */
    protected Object WEIGHT;
    /** Constructor that sets the weight attribute. */
    public MyDijkstra(Object weight_attribute) {
        WEIGHT = weight_attribute;
    }
    /** The edge weight is stored in attribute WEIGHT of the edge. */
    public int weight(Edge e) {
        return ((Integer) e.get(WEIGHT)).intValue();
    }
}

```

375

7.5 习题

基础题

- R-7.1 画出一个具有8个顶点、16条边的简单连通加权图，每条边上有唯一的边权值。确定一个“起始”顶点，并用此图说明Dijkstra算法的执行过程。
- R-7.2 说明如何修改Dijkstra算法，使之适合于有向图的情况。计算从某个源点到其他所有顶点的最短有向路径（directed path）。
- R-7.3 说明如何修改Dijkstra算法，使之不仅输出从 v 到 G 中每个顶点的距离，而且输出根为 v 的树 T ，满足 T 中从 v 到顶点 u 的路径实际上是 G 中从 v 到顶点 u 的最短路径。
- R-7.4 画出一个具有10个顶点、18条边的（简单）有向加权图 G ，使得 G 包含一个至少有四条边的最小权值回路。证明Bellman-Ford算法能找到这个回路。
- R-7.5 算法7-4中的动态规划算法利用空间 $O(n^3)$ 。描述利用 $O(n^2)$ 空间的该算法的一个版本。
- R-7.6 算法7-4中的动态规划算法只计算最短路径距离，而不是实际路径。描述该算法的一个版本，输出有向图中每一对顶点之间的所有最短路径集合。你所设计算法的运行时间应该仍为 $O(n^3)$ 。
- R-7.7 画出一个具有8个顶点、16条边的简单连通无向加权图，每边上有唯一的权值。并用此图说明Kruskal算法的执行过程（注意这个图只有唯一一棵最小生成树）。
- R-7.8 对于Prim-Jarnik算法，重做上一个问题。
- R-7.9 对于Barůvka算法，重做上一个问题。
- R-7.10 考虑用无序序列实现Dijkstra算法中使用的优先队列 Q 。在这种情况下，对于具有 n 个顶点的图，Dijkstra算法最好情况下的运行时间是多少？
提示：考虑每次提取最小元素时 Q 的大小。
- R-7.11 对于图7-2和图7-3中说明的Dijkstra算法的执行过程，描述其图形上约定的意义。箭头意指什么？粗线和虚线又意指什么？
- R-7.12 对于图7-10和图7-12中说明的Kruskal算法的执行过程，重做习题R-7.11。
- 376 R-7.13 对于图7-13和图7-14中说明的Prim-Jarnik算法的执行过程，重做习题R-7.11。
- R-7.14 对于图7-15中说明的Barůvka算法的执行过程，重做习题R-7.11。

创新题

- C-7.1 给出一个具有 n 个顶点的简单图 G 的例子，当用堆实现优先队列时，使得Dijkstra算法的运行时间为 $\Omega(n^2 \log n)$ 。
- C-7.2 给出一个加权有向图 \vec{G} 的例子，图中带有负权值边，但不存在负权值回路，使得Dijkstra算法不正确地计算从某个起始顶点 v 开始的最短路径距离。
- C-7.3 给定一个连通图，找出一条从顶点 $start$ 到顶点 $goal$ 的最短路径，考虑以下贪心策略。
1: 初始化路径为 $start$ 。
2: 初始化 $VisitedVertices$ 为 $\{start\}$ 。
3: 如果 $start = goal$ ，返回 $path$ 并退出。否则，继续。
4: 找最小权值边 $(start, v)$ ，使得 v 与 $start$ 相邻，且 v 不在 $VisitedVertices$ 中。
5: 把 v 添加到 $path$ 中。
6: 把 v 添加到 $VisitedVertices$ 中。
7: 设 $start$ 等于 v ，转到第3步。
这个贪心策略总会找到从 $start$ 到 $goal$ 的一条最短路径吗？直观地解释为什么？或者给出一个反例。
- C-7.4* 给定具有 n 个顶点、 m 条边的加权图 G ，使得每条边上的权值为介于 $0 \sim n$ 之间的一个整数。证明可用 $O(n \log^* n)$ 时间找出 G 的一棵最小生成树。
- C-7.5 证明如果一个连通加权图 G 中的所有权值互不相同，那么存在 G 的唯一一棵最小生成树。
- C-7.6 给定无环加权有向图 \vec{G} ，设计一个找出从顶点 s 到顶点 t 的最长（longest）有向路径的有效算法。详

细说明所用的图表示方式及使用的任何辅助数据结构。同时,分析你所设计算法的时间复杂度。

C-7.7 给定一个电话网络图 G ,其顶点表示交换局,边表示两个交换局之间的通信线路。边上标记它们的带宽。路径的带宽定义为其上最小带宽边上的带宽。已知这个电话网络图及两个交换局 a 和 b ,给出一个算法,输出 a 和 b 之间的一条路径上的最大带宽。

C-7.8 NASA想要利用通信信道连接分布在国家内的 n 个站。每对站之间有不同的带宽可用,称之为优先级。NASA希望选择 $n-1$ 条信道(可能最少),使得信道能够连接所有站,且使总带宽(定义为各个信道带宽之和)达到最大。给出这个问题的一个有效算法,确定它在最坏情况下的时间复杂度。考虑加权图 $G=(V,E)$,其中 V 是站集合, E 是站之间的信道集合。定义边 $e \in E$ 上的权值 $w(e)$ 为相应信道的带宽。

377

C-7.9 已知一个时刻表(timetable)由以下部分组成:

- n 个机场的集合 \mathcal{A} ,以及对于每个机场 $a \in \mathcal{A}$,有最短连接时间 $c(a)$ 。
- m 个航班的集合 \mathcal{F} ,对于每个航班 $f \in \mathcal{F}$:
 - 起始机场 $a_1(f) \in \mathcal{A}$
 - 目的机场 $a_2(f) \in \mathcal{A}$
 - 离开时间 $t_1(f)$
 - 到达时间 $t_2(f)$

描述一个航班调度问题的有效算法。在这个问题中,给定机场 a 和 b ,及一个时间 t ,希望计算航班的一个序列,在时刻 t 或者之后离开机场 a ,使到达机场 b 的时间最早。应该看到使中转机场的连接时间达到最短。利用 n 和 m 的函数,表示你所设计算法的运行时间。

C-7.10 为了把Bigfunnia王国从邪恶的妖怪即“指数渐近”中拯救出来,作为你的报酬,国王给了你一个挣大钱的机会。在城堡的后面,有一个迷宫,沿着迷宫的走廊有许多袋金币。每袋中的金币数量不等。你得到在迷宫中漫游的机会,以找出所有金子。你只能从标记为“ENTER”的入口进入,并从标记为“EXIT”的出口出去(这些是不同的门)。在迷宫中,你不能往回走。迷宫的每个走廊的墙上有一个着色箭头,你只能沿着箭头的方向走下去。没有一条路会遍历迷宫中的“循环”。你会得到一张包括金子数量和每条走廊方向的迷宫图。描述一个算法,帮助你找到最多的金子。

C-7.11 给定具有 n 个顶点的有向图 \vec{G} ,设 M 是对应于 \vec{G} 的 $n \times n$ 邻接矩阵。

a. 设 M 自乘的乘积 M^2 定义如下,对于 $1 \leq i, j \leq n$:

$$M^2(i, j) = M(i, 1) \odot M(1, j) \oplus \cdots \oplus M(i, n) \odot M(n, j)$$

其中“ \oplus ”表示布尔型or运算符,“ \odot ”表示布尔型and运算符。给定这个定义,若 $M^2(i, j) = 1$,蕴涵着顶点 i 和 j 之间有什么关系?若 $M^2(i, j) = 0$,又蕴涵着顶点 i 和 j 之间有什么关系?

b. 假定 M^k 是 M^2 自乘的乘积。 M^k 中的元素表示什么? $M^k = (M^k)$ 中的元素又表示什么?一般而言,矩阵 M^k 中包含什么信息?

c. 假定 \vec{G} 是加权有向图,做出以下假设:

- 1: 对于 $1 \leq i \leq n$, $M(i, i) = 0$ 。
- 2: 对于 $1 \leq i, j \leq n$,如果 $(i, j) \in E$,那么 $M(i, j) = \text{weight}(i, j)$ 。
- 3: 对于 $1 \leq i, j \leq n$,如果 $(i, j) \notin E$,那么 $M(i, j) = \infty$ 。

同时,设 M^2 定义如下,对于 $1 \leq i, j \leq n$:

$$M^2(i, j) = \min\{M(i, 1) + M(1, j), \dots, M(i, n) + M(n, j)\}$$

378

如果 $M^2(i, j) = k$,可得出顶点 i 和 j 之间有什么关系?

C-7.12 说明如何修改Barůvka算法,使算法最坏情况下的运行时间为 $O(n^2)$ 。

程序设计

P-7.1 假设边上权值为整数,实现Kruskal算法。

P-7.2 假设边上权值为整数,实现Prim-Jarník算法。

P-7.3 假设边上权值为整数,实现Barůvka算法。

P-7.4 对于本章讨论的最小生成树算法（Kruskal算法、Prim-Jarník算法或Barůvka算法）中的两个算法进行实验性比较。利用随机生成的图，开发一组广泛的实验，测试这些算法的运行时间。

7.6 本章注记

Barůvka提出了第一个众所周知的最小生成树算法[22]，它发表于1926年。Prim-Jarník算法于1930年首次由Jarník[108]在捷克发表，1957年由Prim[169]在英国发表。1956年Kruskal发表了最小生成树算法[127]。对最小生成树历史的进一步研究感兴趣的读者可参阅Graham和Hell的文章[89]。当前渐近最快的最小生成树算法是Karger、Klein和Tarjan[112]提出的随机方法，其期望运行时间为 $O(m)$ 。

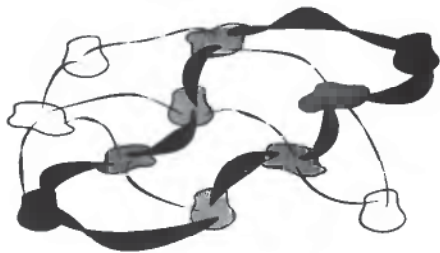
1959年Dijkstra[60]发表了他的单源点最短路径算法。Bellman-Ford算法源于Bellman[25]和Ford[71]各自的出版物。

对图论算法感兴趣的读者，可进一步参阅Ahuja、Magnanti和Orlin[9]，Cormen、Leiserson和Rivest[55]、Even[68]、Gibbons[77]、Mehlhorn[149]、Tarjan[200]以及van Leeuwen[205]的著作。

另外，对于Prim-Jarník算法的运行时间即Dijkstra算法的运行时间，在利用两个更复杂的数据结构之一实现队列 Q 后，实际上可以改进到 $O(n \log n + m)$ 。这两个复杂的数据结构是“斐波纳契堆”[72]或者“可松弛堆”[61]。对这些实现感兴趣的读者可参阅描述这些结构实现的论文，以及它们是如何被应用于最短路径问题和最小生成树问题的。

第8章

网络流和匹配



与加权图有关的一个重要问题是最大流问题 (maximum flow)。在这个问题中, 给定一个加权有向图 G , 其中边表示能够运输某些商品的“管道”。边上的权值表示它能运输的最大量。最大流问题是指从某个顶点 s 开始, 找出一种最大量地运输给定商品到顶点 t 的方法, 其中称顶点 s 为源点 (source), 顶点 t 为汇点 (sink)。

示例8.1 考虑用有向图 G 模型化的部分因特网, 其中每个顶点表示一台计算机, 每条边 (u, v) 表示一条从计算机 u 到计算机 v 的单向通信信道, 每条边 (u, v) 上的权值表示信道的带宽, 即一秒钟内从 u 向 v 所能发送的最大字节数。如果想要从 G 中的某台计算机 s 向 G 中的某台计算机 t 发送高带宽的流媒体连接, 那么发送该连接的最快方式是将其分成若干个数据包, 并按照最大流通过 G 路由这些包 (如图8-1所示)。

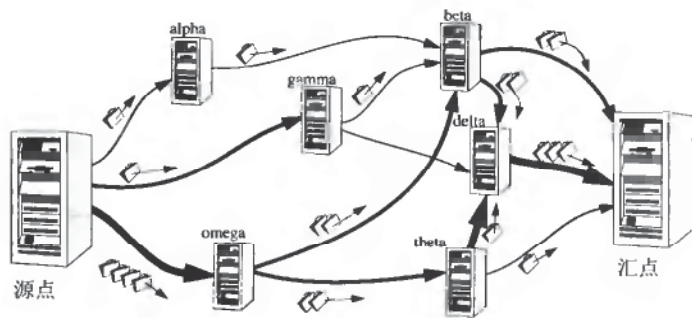


图8-1 表示计算机网络的图中的流示例, 粗边表示的带宽为4 MB/s, 中等粗细的边表示的带宽为2 MB/s, 细边表示的带宽为1 MB/s。用文件夹图标表示一条边上发送数据的量, 每个文件夹与通过该信道的1 MB/s数据量相对应。注意, 从源点发送到汇点的总流量 (6 MB/s) 不是最大的。的确, 还可以另外发送1 MB/s的数据量, 从源点到 γ , 再到 δ , 最后到汇点。添加这个额外的流后, 总流量达到最大

382

最大流问题与找出图中一种类型的顶点与另一种类型的顶点匹配的最大方式密切相关。因而, 我们也研究最大匹配问题, 并表明如何有效解决最大流问题。

有时, 会碰到许多不同的最大流。尽管就产生流的多少而言, 它们都是最大流, 但事实上这些流可能在它们的代价上有所不同。因此, 在这一章里, 还研究计算具有最小代价的最大流的方法。当有许多不同的最大流时, 则会有某种测量其相对代价的方式。本章的结尾是用Java实现一

个最小代价流的算法。

8.1 流和割

上述例子说明了一个合法流必须遵循的规则。为了精确地描述出这些规则,先定义流的概念。

8.1.1 流网络

流网络 (flow network) N 由以下几部分组成:

- 有向连通图 G , 其边上权值为非负整数, 其中边 e 上的权值称为 e 的容量 (capacity) $c(e)$ 。
- G 中两个不同的顶点 s 和 t , 分别称为源点 (source) 和汇点 (sink), 满足 s 没有入边, t 没有出边。

给定这样一个标记图, 一个挑战性的问题是, 确定可从 s 到 t 的某些商品的流量, 使得经过那条边的最大流满足边的容量约束条件 (如图8-2所示)。

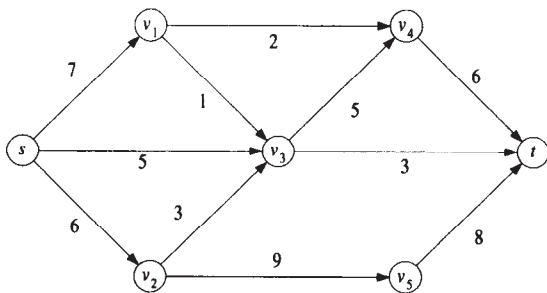


图8-2 流网络 N 。 N 中的每条边标记有它的容量 $c(e)$

当然, 如果希望某些商品从 s 流到 t , 需要更精确地定义什么是“流”。网络 N 的流 (flow) 是给 G 中的每条边 e 赋予的整数值 $f(e)$, 满足以下性质:

- 对于 G 中的每条边 e ,

$$0 \leq f(e) \leq c(e) \quad (\text{容量定律})$$

- 对于 G 中不同于源点 s 和汇点 t 的每个顶点 v ,

$$\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e) \quad (\text{守恒定律})$$

其中 $E^-(v)$ 和 $E^+(v)$ 分别表示 v 的入边的集合和出边的集合。

换句话说, 一个流必须满足边容量约束条件, 并且满足对于不同于 s 和 t 的每个顶点 v , 流出 v 的总流量等于流入 v 的总流量。例如, 图8-3中对于流的说明满足上述两条定律。

称 $f(e)$ 为边 e 上的流 (flow)。流 f 的值 (value) 表示为 $|f|$, 等于从源点 s 流出的总流量:

$$|f| = \sum_{e \in E^+(s)} f(e)$$

容易证明, 流的值也等于进入汇点 t 的总流量 (见习题R-8.1):

$$|f| = \sum_{e \in E^-(t)} f(e)$$

即一个流确定了某些商品如何从 s 流出, 通过网络 N , 最终流入汇点 t 。流网络 N 的最大流 (maximum flow) 是一个具有 N 上所有流的最大值的流 (如图8-4所示)。由于最大流可以最有效地使用流网

络, 因而, 我们对计算最大流的方法最感兴趣。

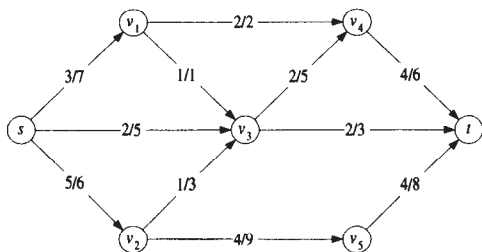


图8-3 图8-2中的流网络 N 的流 f (其值 $|f| = 10$)

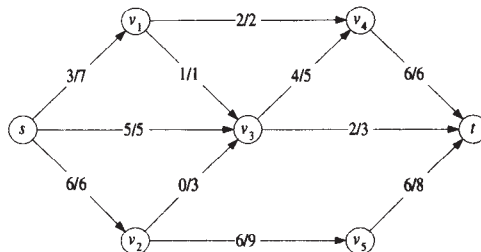


图8-4 图8-2中的流网络 N 的最大流 f^* (其值 $|f^*| = 14$)

384

8.1.2 割

由此可得, 流与另一个称为割的概念关系密切。直观上讲, 割用于把流网络 N 中的顶点分成两个集合 s 和 t 。形式上, N 的割 (cut) 是 N 中顶点的一个划分 $\chi = (V_s, V_t)$, 满足 $s \in V_s$ 且 $t \in V_t$ 。称 N 中源点为 $u \in V_s$ 、目的点为 $v \in V_t$ 的边 e 为割 χ 的前向边 (forward edge)。称源点在 V_t 中, 目的点在 V_s 中的边 e 为后向边 (backward edge)。把割看作对 N 中的边做一切割, 使 s 和 t 分离, 其前向边从 s 一侧到 t 一侧, 后向边则在相反的方向上 (如图8-5所示)。

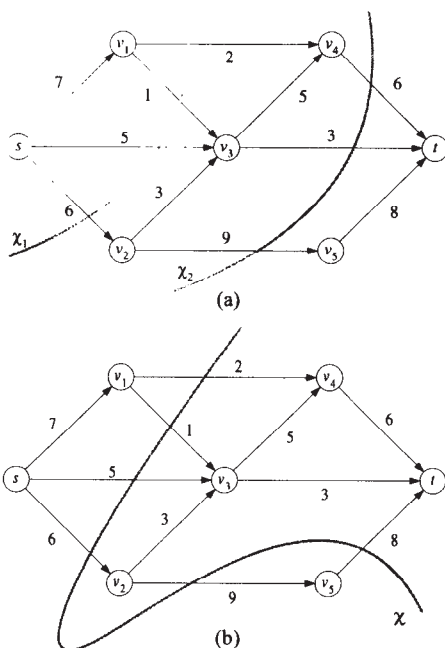


图8-5 (a)图8-2的流网络 N 中的两个割 χ_1 (左边) 和 χ_2 (右边), 这些割只有前向边, 它们的容量分别是 $c(\chi_1) = 14$ 和 $c(\chi_2) = 18$, 割 χ_1 是 N 的最小割; (b) N 中既有前向边又有后向边的割 χ , 它的容量为 $c(\chi) = 22$

385

给定 N 中的一个流 (穿越割 χ 的流, 用 $f(\chi)$ 表示), 等于 χ 中前向边的流之和减去后向边的流之和。即 $f(\chi)$ 是从 χ 中 s 一侧流到 χ 中 t 一侧的商品的净流量。以下引理给出了 $f(\chi)$ 的一个有趣的性质。

引理8.1 设 N 是一个流网络, f 是 N 的一个流。对于 N 的任何割 χ , f 的值等于穿越割 χ 的流, 即

$|f| = f(\chi)$ 。

证明 考虑下列求和式

$$F = \sum_{v \in V_s} \left(\sum_{e \in E^+(v)} f(e) - \sum_{e \in E^-(v)} f(e) \right)$$

根据守恒定律, 对于 V_s 中不同于 s 的每个顶点 v , 可得 $\sum_{e \in E^+(v)} f(e) - \sum_{e \in E^-(v)} f(e) = 0$ 。因此, $F = |f|$ 。

另一方面, 对于既不是割 χ 中前向边也不是后向边的每条边 e , 和 F 包含项 $f(e)$ 和项 $-f(e)$, 它们相互抵消, 因此, $F = f(\chi)$ 。■

上述定理表明, 不论在什么地方切割流网络, 分隔出 s 和 t , 穿越那个割的流总是等于整个网络的流。割 χ 的容量 (capacity) 等于 χ 中前向边上的容量之和 (注意, 没有包含后向边), 用 $c(\chi)$ 表示, 下一个引理表明, 割容量 $c(\chi)$ 是穿越 χ 的任何流的上界。

引理8.2 设 N 是一个流网络, χ 是 N 的一个割。给定 N 的任何流 f , 穿越割 χ 的流不会超过割 χ 的容量, 即 $f(\chi) \leq c(\chi)$ 。

证明 用 $E^+(\chi)$ 表示 χ 中的前向边, 用 $E^-(\chi)$ 表示 χ 中的后向边。由 $f(\chi)$ 的定义, 可得

$$f(\chi) = \sum_{e \in E^+(\chi)} f(e) - \sum_{e \in E^-(\chi)} f(e)$$

从上述求和中去掉非正项, 可得 $f(\chi) \leq \sum_{e \in E^+(\chi)} f(e)$ 。由容量定律可知, 对于每条边 e , $f(e) \leq c(e)$ 。因此, 可得

$$f(\chi) \leq \sum_{e \in E^+(\chi)} c(e) = c(\chi) \quad \blacksquare$$

结合引理8.1和引理8.2, 得到以下关于流和割的重要结论。

定理8.1 设 N 是一个流网络。给定 N 的任意流 f 以及 N 的任意割 χ , f 的值不会超过割 χ 的容量, 即 $|f| \leq c(\chi)$ 。

换句话说, 给定流网络 N 的任意割 χ , χ 的容量是 N 的任意流的上界。这个上界甚至对于 N 的最小割 (minimum cut) 也成立。最小割是 N 的所有割中具有最小容量的割。在图8-5的例子中, χ_1 是一个最小割。

386

8.2 最大流

定理8.1蕴涵着最大流的值不超过最小割的容量。在这一节里将要证明这两个量实际上是相等的。在论述过程中, 将描述构造最大流的一种方法。

8.2.1 剩余容量和增大路径

为了证明某个流 f 最大, 需要通过一种方法证明绝对没有更多的流可以“挤进” f 中。利用以下讨论的剩余容量和增大路径的相关概念, 恰好可以给出 f 是最大流的一个证明。

1. 剩余容量

设 N 是图 G 指定的一个流网络, 其容量函数为 c , 源点为 s , 汇点为 t 。此外, 设 f 是 N 的一个流。给定 G 中从顶点 u 到顶点 v 方向上的一条边 e , 关于流 f 的从 u 到 v 的剩余容量 (residual capacity) 用 $\Delta_f(u, v)$ 表示, 定义为

$$\Delta_f(u, v) = c(e) - f(e)$$

从 v 到 u 的剩余容量则定义为

$$\Delta_f(v, u) = f(e)$$

直观上讲, 流 f 所定义的剩余容量是从 s 向 t “推进”流的过程中, f 尚未完全利用的额外容量。

设 π 是从 s 到 t 的一条路径, 在这条路径上可以在前向边或是后向边的方向上进行遍历, 即可以遍历从源点 u 到目的点 v 或是从目的点 v 到源点 u 上的边 $e(u, v)$ 。形式上, π 的一条前向边(forward edge)是 π 中的一条边, 满足沿着 π 从 s 到 t 的行进过程中, 将会在 e 的目的点之前遇见 e 的源点。称 π 中不是前向边的边为后向边(backward edge)。我们把剩余容量的定义扩展到 π 中从 u 到 v 遍历的边 e 上, 满足 $\Delta_f(e) = \Delta_f(u, v)$ 。换句话说,

$$\Delta_f(e) = \begin{cases} c(e) - f(e) & \text{如果 } e \text{ 是一条前向边} \\ f(e) & \text{如果 } e \text{ 是一条后向边} \end{cases}$$

即行进在边 e 的前向方向上的剩余容量是 f 正要消耗的 e 的额外容量, 但是, 在相反方向上的剩余容量则是 f 已经消耗的流(且可能潜在地“归还”, 如果允许另一个流具有更高的值)。

387

2. 增大路径

路径 π 上的剩余容量 $\Delta_f(\pi)$ 定义为该路径所在边上的最小剩余容量, 即

$$\Delta_f(\pi) = \min_{e \in \pi} \Delta_f(e)$$

这个值是我们可能沿着 π 推进的额外流的最大量, 同时不违反容量约束条件。流 f 的增大路径(augmenting path)是一条从源点 s 到汇点 t 且具有非零剩余容量的路径 π , 即对于 π 中的每条边 e 。

- $f(e) < c(e)$, 如果 e 是一条前向边。
- $f(e) > 0$, 如果 e 是一条后向边。

图8-6中显示了一个增大路径的例子。

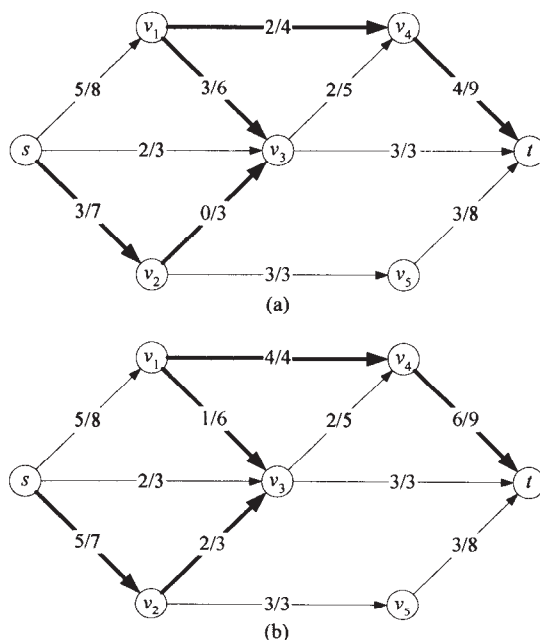


图8-6 增大路径的示例: (a)网络 N , 流 f , 粗边表示增大路径 π (v_1, v_3)是一条后向边); (b)沿着从 s 到 t 的路径 π , 在流 f 中推进 $\Delta_f(\pi) = 2$ 个单位的流, 得到流 f'

388

正如以下引理所示,总是可以把增大路径上的剩余容量添加到一个现有的流中,得到另一个有效的流。

引理8.3 设 π 是网络 N 中流 f 的一条增大路径。存在 N 的流 f' ,其值为 $|f'| = |f| + \Delta_f(\pi)$ 。

证明 修改 π 中边上的流,来计算流 f' :

$$f'(e) = \begin{cases} f(e) + \Delta_f(\pi) & \text{如果 } e \text{ 是一条前向边} \\ f(e) - \Delta_f(\pi) & \text{如果 } e \text{ 是一条后向边} \end{cases}$$

注意:如果 e 是一条后向边,则减去 $\Delta_f(\pi)$ 。在这种情况下,要减去 f 已经取走的边 e 上的流。在任何情况下,由于 $\Delta_f(\pi) \geq 0$ 是 π 中任何边上的最小剩余容量,加上 $\Delta_f(\pi)$ 之后不会违反前向边上的容量约束条件,减去 $\Delta_f(\pi)$ 之后则不会使任何后向边上的流为负。因此, f' 是 N 的有效流,且 f' 值为 $|f| + \Delta_f(\pi)$ 。 ■

由引理8.3可知,对于一个流 f ,若存在一条增大路径 π ,则蕴涵着 f 不是最大流。同样,给定一条增大路径 π ,沿着从 s 到 t 的路径 π ,通过推进 $\Delta_f(\pi)$ 个单位的流,改变 f 的值,使其值增加。如引理8.3的证明。

如果网络 N 中不存在流 f 的一条增大路径,则会如何?在这种情况下,可得 f 是一个最大流,正如以下引理所阐述的那样。

引理8.4 如果网络 N 中不存在关于流 f 的增大路径,那么 f 是一个最大流。同时,存在 N 的一个割 χ ,满足 $|f| = c(\chi)$ 。

证明 设 f 是 N 的一个流,假定 N 中不存在关于流 f 的增大路径。可以由 f 构造一个割 $\chi = (V_s, V_t)$,其方式是:把所有顶点 v 放入 V_s 中,使得从源点 s 到顶点 v 的路径上的边由那些非零剩余容量的边组成。这样一条路径称为从 s 到 v 的增大路径。集合 V_t 包含 N 中其余的顶点。因为不存在流 f 的增大路径,所以 N 的汇点 t 在 V_t 中。因此, $\chi = (V_s, V_t)$ 满足割的定义。

由 χ 的定义可知,割 χ 的每条前向边和后向边上的剩余容量为0,即

$$f(e) = \begin{cases} c(e) & \text{如果 } e \text{ 是 } \chi \text{ 的一条前向边} \\ 0 & \text{如果 } e \text{ 是 } \chi \text{ 的一条后向边} \end{cases}$$

因此, χ 的容量等于 f 的值,即

$$|f| = c(\chi)$$

由定理8.1可知, f 是最大流。 ■

由定理8.1和引理8.4可得以下关于最大流和最小割的基本结论。

389

定理8.2 (最大流、最小割定理) 最大流的值等于最小割的容量。

8.2.2 Ford-Fulkerson 算法

Ford和Fulkerson提出的经典算法用于计算网络中的最大流,其方法是:把贪心法应用到用于证明最大流、最小割定理(定理8.2)的增大路径方法上。

Ford-Fulkerson 算法(Ford-Fulkerson algorithm)的主要思想是分阶段逐渐增大一个流的值,在每一个阶段中,沿着一条从源点到汇点的增大路径推进一定量的流。起初,每条边上的流为0,在每一个阶段,计算一条增大路径 π ,并沿着 π 推进等于 π 的剩余容量的流量,如在引理8.3中所证明的那样。在当前流 f 不允许增大路径时,算法终止。在这种情况下,引理8.4

保证 f 是最大流。

算法8-1中给出了Ford-Fulkerson算法找出最大流问题解的伪代码描述。

算法8-1 计算网络中最大流的Ford-Fulkerson算法

```

算法 MaxFlowFordFulkerson( $N$ ):
  输入: 流网络  $N = (G, c, s, t)$ 
  输出:  $N$  的最大流  $f$ 
  for 每条边  $e \in N$  do
     $f(e) \leftarrow 0$ 
   $stop \leftarrow \text{false}$ 
  repeat
    从  $s$  开始遍历  $G$ , 找出  $f$  的一条增大路径
  if 存在一条增大路径  $\pi$  then
    {计算  $\pi$  的剩余容量  $\Delta_f(\pi)$ }
     $\Delta \leftarrow +\infty$ 
    for 每条边  $e \in \pi$  do
      if  $\Delta_f(e) < \Delta$  then
         $\Delta \leftarrow \Delta_f(e)$ 
    {沿着路径  $\pi$  推进  $\Delta = \Delta_f(\pi)$  个单位的流}
    for 每条边  $e \in \pi$  do
      if  $e$  是一条前向边 then
         $f(e) \leftarrow f(e) + \Delta$ 
      else
         $f(e) \leftarrow f(e) - \Delta$  { $e$  是一条后向边}
    else
       $stop \leftarrow \text{true}$  { $f$  是一个最大流}
  until  $stop$ 

```

图8-7对Ford-Fulkerson算法进行了可视化。

390

实现细节

Ford-Fulkerson算法有许多重要的实现细节，这些细节影响了表示流和计算增大路径的方式。实际上表示一个流相当简单。可以把网络的每条边标记一个属性，表示沿着所在边的流（6.5节）。为了计算一条增大路径，对表示流网络的图 G 进行特殊遍历。这样的遍历是对DFS遍历（6.3.1节）或是BFS遍历（6.3.3节）的简单修改，这里没有考虑依附当前顶点 v 的所有边，而只考虑以下边：

- v 的那些流小于容量的出边
- v 的那些具有非零流的入边

另外，计算关于当前流 f 的增大路径可以归约为在由 G 导出的新有向图 R_f 中找出一条简单路径的问题。 R_f 中的顶点与 G 中的顶点相同。对于 G 中相邻顶点 u 和 v 的每个有序对，如果 $\Delta_f(u, v) > 0$ ，则添加一条从 u 到 v 的有向边。称图 R_f 为关于流 f 的剩余图（residual graph）。关于流 f 的一条增大路径对应于剩余图 R_f 中从 s 到 t 的一条有向路径。可以通过从源点 s 开始对图 R_f 进行DFS遍历，来计算这条路径（见6.3节和6.5节）。

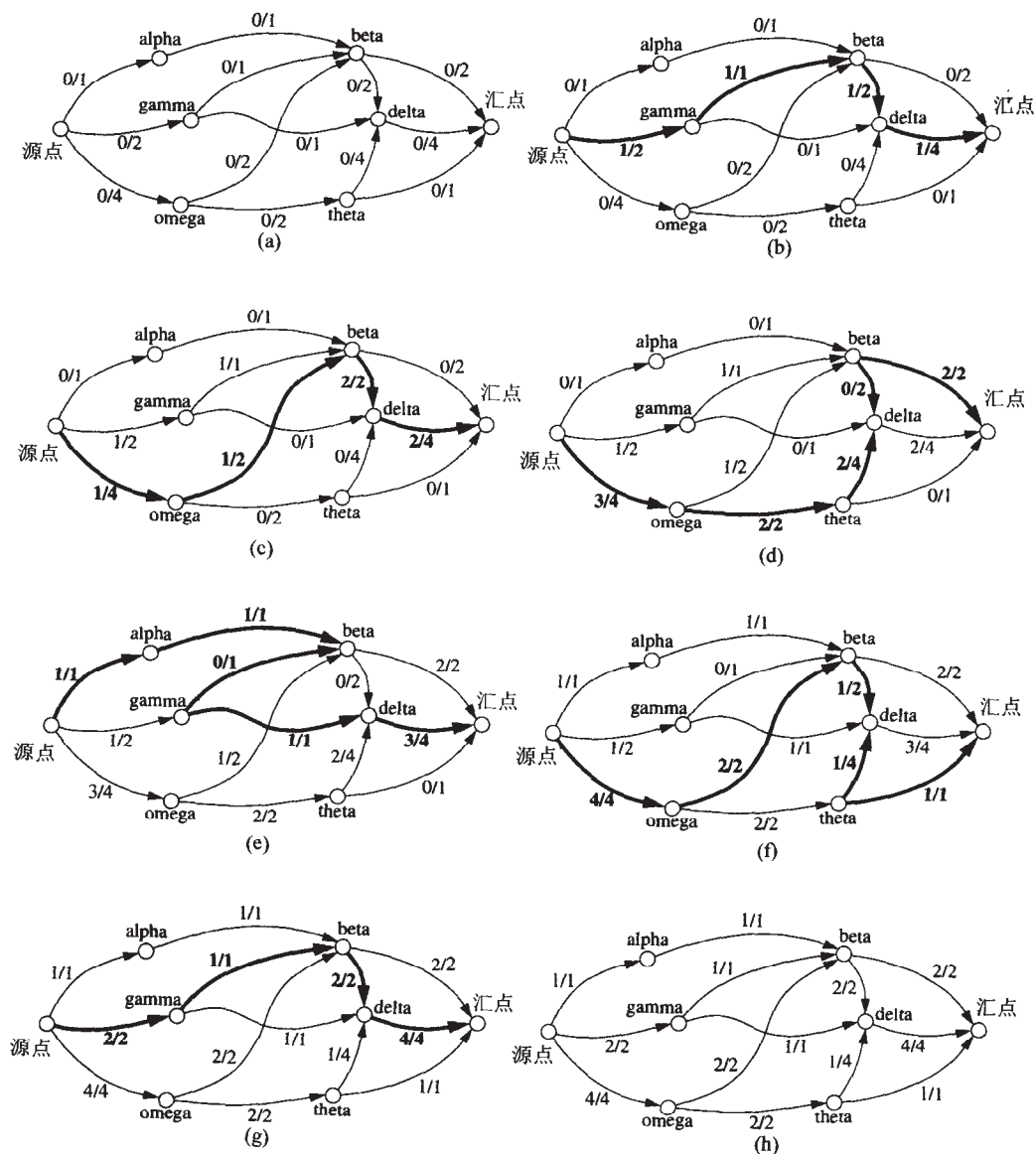


图8-7 Ford-Fulkerson算法在图8-1的流网络上的执行示例。增人路径用粗线画出

8.2.3 Ford-Fulkerson 算法分析

分析Ford-Fulkerson算法的运行时间需要一点技巧。这是因为算法并没有指定查找增大路径的准确方式，正如将要看到的那样，增大路径的选择对算法的运行时间有着重大影响。

设 n 和 m 分别为流网络中的顶点数和边数， f^* 是一个最大流。因为表示网络的图是连通的，则有 $n \leq m + 1$ 。注意，每当找到一条增大路径，就把流的值至少增加1，这是因为边的容量和流是整数。因此，最大流的值 $|f^*|$ 是算法查找最大路径次数的一个上界。同时注意，可以用简单图遍历方法，如DFS遍历或BFS遍历，找到一条增大路径，所需时间为 $O(m)$ （见定理6.6和定理6.9，且 $n \leq m + 1$ ）。因此，Ford-Fulkerson算法的运行时间可以限制为至多 $O(|f^*| \cdot m)$ 。如图8-8中的说明，通过选择某些增大路径，实际上可以达到这个界限。由此可知，Ford-Fulkerson算法是一个伪多

项式时间算法(5.3.3节), 因为它的运行时间取决于输入大小和数值参数的值。因此, 如果 $|f^*|$ 较大, 且增大路径选择得不好, 那么Ford-Fulkerson算法的运行时间界限可能相当慢。

391
392

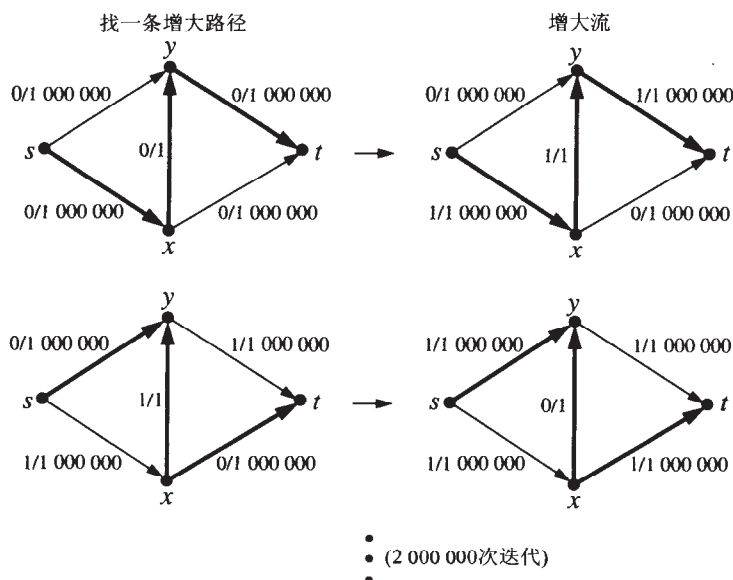


图8-8 标准Ford-Fulkerson算法缓慢运行的网络例子。如果算法交替选择 (s, x, y, t) 和 (s, y, x, t) 作为增大路径, 那么算法总共会执行2 000 000次迭代, 尽管只需要两次迭代就足够了

8.2.4 Edmonds-Karp 算法

Edmonds-Karp算法(Edmonds-Karp algorithm)是Ford-Fulkerson算法的一个变体。它利用一种简单的技术来找出良好的增大路径, 加快算法的运行速度。这项技术基于“更贪心”的概念, 把贪心法应用于最大流问题。也就是说, 在每次迭代中, 选择一条具有最少边数的路径, 这可以通过修改BFS遍历过程, 在 $O(m)$ 时间内就可容易做到。可以证明利用这些Edmonds-Karp增大路径(Edmonds-Karp augmentation), 迭代次数不会超过 nm , 这蕴涵着Edmonds-Karp算法的运行时间为 $O(nm^2)$ 。

首先引入一些符号。称路径 π 的长度(length)为 π 中的边数。设 f 是网络 N 的一个流。给定一个顶点 v , 用 $d_f(v)$ 表示从源点 s 到顶点 v 的关于 f 的一条增大路径的最短长度, 称这个量为 v 关于流 f 的剩余距离(residual distance)。

以下讨论说明了每个顶点的剩余距离是如何影响Edmonds-Karp算法的运行时间的。

393

Edmonds-Karp算法的性能

在开始分析时, 首先注意到剩余距离是Edmonds-Karp增大路径序列上的一个非降函数。

引理8.5 设 g 是流 f 沿着最短长度的路径 π 增大时得到的流。那么对于每个顶点 v ,

$$d_f(v) \leq d_g(v)$$

证明 假定存在某个顶点违反上述不等式。设 v 是关于 g 且有最小剩余距离的这样一个顶点, 即

$$d_f(v) > d_g(v) \quad (8.1)$$

且

$$d_g(v) \leq d_g(u), \text{ 对于每个 } u \text{ 满足 } d_f(u) > d_g(u) \quad (8.2)$$

考虑从 s 到 v 的关于流 g 的最短长度的一条增大路径 γ 。设 u 是 γ 上顶点 v 的直接前驱, e 是 γ 上两个端点分别为 u 和 v 的边 (见图8-9)。由上述定义可得,

$$\Delta_g(u, v) > 0 \quad (8.3)$$

同样, 因为 u 是最短路径 γ 上 v 的直接前驱, 可得

$$d_g(v) = d_g(u) + 1 \quad (8.4)$$

最后, 由公式(8.2)和公式(8.4) 可得,

$$d_f(u) \leq d_g(u) \quad (8.5)$$

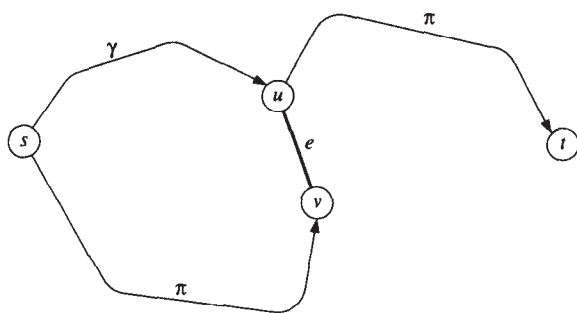


图8-9 引理8.5的证明的说明

现在证明 $\Delta_f(u, v) = 0$ 。实际上, 如果有 $\Delta_f(u, v) > 0$, 就能从 u 到 v 沿着关于流 f 的增大路径行进。这蕴涵着

$$\begin{aligned} d_f(v) &\leq d_f(u) + 1 \\ &\leq d_g(u) + 1 \text{ 根据公式(8.5)} \\ &= d_g(v) \text{ 根据公式(8.4)} \end{aligned}$$

这与公式(8.1)矛盾。

因为 $\Delta_f(u, v) = 0$, 且由公式(8.3)可知, $\Delta_g(u, v) > 0$, 增大路径 π 从 f 产生 g , 从而必定遍历了从 v 到 u 的边 e (见图8-9)。因此,

$$\begin{aligned} d_f(v) &= d_f(u) - 1 \text{ 因为 } \pi \text{ 是一条最短路径} \\ &\leq d_g(u) - 1 \text{ 根据公式(8.5)} \\ &\leq d_g(v) - 2 \text{ 根据公式(8.4)} \\ &< d_g(v) \end{aligned}$$

因此, 得到与公式(8.1)矛盾的结论。这就完成了定理的证明。 ■

394

直观上讲, 引理8.5蕴涵着: 每次进行Edmonds-Karp增大时, 从 s 到任意顶点 v 的剩余距离只能增大或者保持不变。这个事实给出以下引理。

引理8.6 在具有 n 个顶点和 m 条边的网络上, 执行Edmonds-Karp算法后, 流增大的数量不会超过 nm 。

证明 设 f_i 是第 i 次增大之前网络中的流, 设 π_i 是这个增大过程中所用的路径。如果 e 的剩余容量等于 π_i 的剩余容量, 则称 π_i 中的一条边 e 是 π_i 的瓶颈 (bottleneck)。显然, Edmonds-Karp算法所

用的每条增大路径至少有一个瓶颈。

考虑边 e 连接的一对顶点 u 和 v ，假定边 e 是两条增大路径 π_i 和 π_k （其中 $i < k$ ）从 u 到 v 遍历 e 的一个瓶颈。上述假设蕴涵以下结论：

- $\Delta_{f_i}(u, v) > 0$
- $\Delta_{f_{i+1}}(u, v) = 0$
- $\Delta_{f_k}(u, v) > 0$

因此，必定有一个中间的第 j 次增大，其中 $i < j < k$ 。它的增大路径 π_j 遍历从 v 到 u 的边 e 。于是可得

$$\begin{aligned} d_f(u) &= d_f(v) + 1 \quad (\text{因为}\pi_j\text{是一条最短路径}) \\ &\geq d_{f_i}(v) + 1 \quad (\text{根据引理8.5}) \\ &\geq d_{f_i}(u) + 2 \quad (\text{因为}\pi_i\text{是一条最短路径}) \end{aligned}$$

由于顶点的剩余距离总是小于顶点数 n ，在Edmonds-Karp算法的执行过程中，每条边可能是瓶颈的次数至多为 n （在增大路径可能遍历的两个方向的每个方向上各为 $n/2$ 次）。因此，增大的总数量不会超过 nm 。■

因为利用修改的BFS策略，可以在 $O(m)$ 时间内完成单一流增大的过程。上述讨论可以概括如下。

定理8.3 给定具有 n 个顶点和 m 条边的流网络，计算最大流的Edmonds-Karp算法的运行时间为 $O(nm^2)$ 。

395

8.3 最大二分匹配

许多重要的应用中出现的一个问题是最大二分匹配（maximum bipartite matching）问题。在这个问题中，给定一个具有如下性质的连通无向图：

- 将 G 中的顶点划分成两个集合 X 和 Y 。
- G 中的每条边上的端点一个在 X 中，另一个在 Y 中。

这样的图称为二分图（bipartite graph）。 G 中的一个匹配（matching）是没有公共端点的边的集合——这样的集合把 X 中的顶点与 Y 中的顶点“配对”，使得每个顶点在另一个集合中至多有一个“伙伴”。最大二分匹配问题是找出一个具有最大边数的匹配（在所有匹配上）。

示例8.2 设 G 是二分图，其中集合 X 表示年轻男士组，集合 Y 表示年轻女士组，他们都出现在社区的舞会上。如果 X 中的 x 和 Y 中的 y 愿意相互成为舞伴，则有一条连接 x 和 y 的边。 G 中的最大匹配对应于可以同时幸福跳舞的男士和女士对的最大集合。

示例8.3 设 G 是二分图，其中集合 X 表示大学课程组，集合 Y 表示教室组。基于注册和声音-视觉的需要，如果在 Y 中的教室 y 讲授 X 中的课程 x ，则有一条连接 x 和 y 的边。 G 中的最大匹配对应于可以同时讲授且不发生冲突的大学课程的最大集合。

这两个例子提供了可以用最大二分匹配问题求解某种应用的一个小示例。幸运的是，存在求解最大二分匹配问题的一种简单方法。

归约到最大流问题

设 G 是二分图，它的顶点被分成两个集合 X 和 Y 。构造一个流网络 H ，使得 H 中的一个最大流

可以直接转换为 G 中的一个最大匹配:

- 首先把 G 中的所有顶点包含在 H 中, 加上一个新的源点 s 和一个新的汇点 t 。
- 接下来, 把 G 中的每一条边添加到 H 中, 但是使得边的方向从 X 中的端点指向 Y 中的端点。此外, 插入一条从 s 到 X 中每个顶点的有向边, 以及插入一条从 Y 中的每个顶点到 t 的有向边。
- 最后, 给 H 中的每一条边赋予容量1。

396

给定 H 的一个流 f , 用 f 定义 G 中边的一个集合 M , 利用如下规则: 只要 $f(e) = 1$, 则边 e 在 M 中 (见图8-10)。现在证明集合 M 是一个匹配。因为 H 中的容量都是1, 通过 H 中每条边的流为0或1。此外, 由于 X 中的每个顶点 x 只有一条入边, 因此守恒定律表明 x 至多有一条出边具有非零流。类似地, 因为 Y 中的每个顶点 y 只有一条出边, 因此 y 至多有一条入边具有非零流。因此, M 把 X 中的每个顶点 x 与 Y 中的至多一个顶点配对, 即集合 M 是一个匹配。同时, 可以容易地看出 M 的大小为 $|f|$, 即流 f 的值。

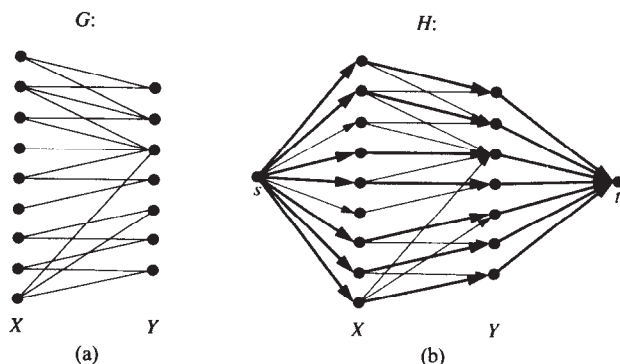


图8-10 (a)二分图 G ; (b)由 G 导出的流网络 H 和 H 中的一个最大流; 粗边上有一个单位的流, 其他边上的流为0

也可定义逆转换, 即给定图 G 中的一个匹配 M , 可以利用以下规则, 用 M 定义 H 的一个流 f :

- 对于 H 中且也在 G 中的每条边 e , 如果 $e \in M$, 则 $f(e) = 1$; 否则, $f(e) = 0$ 。
- 对于 H 中依附于 s 或 t 的每条边 e , 如果 v 是 M 中某条边的端点, 那么 $f(e) = 1$; 否则, $f(e) = 0$, 其中 v 表示 e 的另一个端点。

容易证明, f 是 H 的一个流, 且 f 的值等于 M 的大小。

于是, 可用任意一个最大流算法求解具有 n 个顶点和 m 条边的图 G 上的最大二分匹配问题。

(1) 由二分图 G 构造一个网络 H 。这一步所需时间为 $O(n+m)$ 。网络 H 中有 $n+2$ 个顶点和 $n+m$ 条边。

(2) 利用标准Ford-Fulkerson算法计算 H 的一个最大流。因为最大流的值等于 $|M|$, 即为最大匹配的大小, 且 $|M| \leq n/2$, 所以这一步所需时间为 $O(n(n+m))$ 。由于 G 是连通的, 因而这个时间为 $O(nm)$ 。

于是, 得到如下定理。

397

定理8.4 设 G 是具有 n 个顶点和 m 条边的二分图。可在 $O(nm)$ 时间内计算 G 中的一个最大匹配。

8.4 最小代价流

最大流问题有另一个变体, 可应用于如下情形: 通过一条边发送一个单位的流关联有一个代价。在这一节里, 通过指定每条边 e 的第二个非负整型权值 $w(e)$ (它表示边 e 的代价 (cost)), 来

扩展网络的定义。

给定一个流 f ，定义 f 的代价为

$$w(f) = \sum_{e \in E} w(e)f(e)$$

其中 E 表示网络中边的集合。如果 f 在值为 $|f|$ 的所有流中具有最小代价，则称流 f 是最小代价流 (minimum-cost flow)。最小代价流问题 (minimum-cost flow problem) 包含在所有最大流上找出具有最小代价的最大流。最小代价流问题的一个变体要求找出具有给定流值的一个最小代价流。给定一条关于流 f 的增大路径 π ，定义 π 的代价为 π 中前向边上的代价之和减去 π 中后向边上的代价之和，用 $w(\pi)$ 表示。

8.4.1 增大回路

一条关于 f 的增大回路 (augmenting cycle) 是一条增大路径，它的第一个顶点和最后一个顶点相同。利用更加数学化的说法，它是一个顶点为 $v_0, v_1, \dots, v_k = v_0$ 的有向回路 γ ，满足 $\Delta_f(v_i, v_{i+1}) > 0$ ，其中 $i = 0, \dots, k-1$ (见图8-11)。剩余容量 (8.2.1节给出) 和代价 (上面给出) 的定义也可应用到一条增大回路上。此外，注意，因为它是一个回路，所以可以把增大回路上的流添加到一个现有流中，而不会改变它的流值。

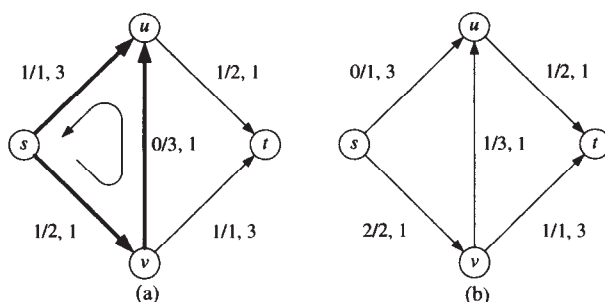


图8-11 (a)具有流 f 的网络，其中每条边 e 标记有 $f(e)/c(e), w(e)$ 。 $|f| = 2$ ， $w(f) = 8$ 。用粗边画出增大回路 $\gamma = (s, v, u, s)$ 。 γ 的剩余容量是 $\Delta_f(\gamma) = 1$ 。 γ 的代价是 $w(\gamma) = -1$ 。(b)从 f 沿着回路 γ 推进一个单位的流得到流 f' 。 $|f'| = |f|$ ，且 $w(f') = w(f) + w(\gamma)\Delta_f(\gamma) = 8 + (-1) \cdot 1 = 7$

398

1. 从增大回路中添加流

以下引理类似于引理8.3，正如它所表明的那样，利用一条增大回路可把一个最大流转换成另一个最大流。

引理8.7 设 γ 是网络 N 中流 f 的一个增大回路。存在 N 的流值为 $|f'| = |f|$ 的流 f' ，且代价为

$$w(f') = w(f) + w(\gamma)\Delta_f(\gamma)$$

引理8.7的证明留作习题 (R-8.13)。

2. 最小代价流的条件

注意：引理8.7蕴涵着如果流 f 有一个负代价的增大回路，那么， f 不存在最小代价。以下定理表明，它的逆命题也为真。这个结论给出测试一个流事实上是否是最小代价流的条件。

定理8.5 当且仅当不存在关于 f 的负代价增大回路时，流 f 才在值为 $|f|$ 的所有流中具有最小代价。

证明 由引理8.7直接得出“仅当”部分成立。为了证明“当”部分，假定流 f 没有最小代价，设 g 是具有最小代价的值为 f 的流。由 f 沿着增大回路通过一系列增大得到流 g 。因为 g 的代

价小于 f 的代价，所以这些回路中必定至少有一个代价为负。 ■

3. 用于找出最小代价流的算法

定理8.5提出了基于沿着负代价回路反复增大流来求解最小代价流问题的一个算法。首先利用Ford-Fulkerson算法或者Edmonds-Karp算法找出一个最大流 f^* 。接下来，确定流 f^* 是否是容许一条负代价的增大回路。可用Bellman-Ford算法（7.1.2节）在 $O(nm)$ 时间内找一条出负回路。设 w^* 表示初始最大流 f^* 的总代价。在每次执行Bellman-Ford算法之后，流的代价至少减少一个单位。因此，从最大流 f^* 开始，可以在 $O(w^*nm)$ 时间内计算具有最小代价的一个最大流。于是，得到以下定理：

定理8.6 给定具有 n 个顶点的流网络 N 和关联其 m 条边的代价，以及一个最大流 f^* ，可在 $O(w^*nm)$ 时间内计算具有最小代价的一个最大流，其中 w^* 是 f^* 的总代价。

399

但是，通过仔细分析如何计算增大回路，我们还可以做得更好。本节余下内容将讨论这个问题。

8.4.2 连续最短路径

在这一节里，提出另一种计算最小代价流的方法。其思想是从一个空流开始，沿着最小代价路径，通过一系列增大过程构造一个最大流。以下定理提供了此方法的基础。

定理8.7 设 f 是一个最小代价流， f' 是沿着具有最小代价的增大路径 π 增大 f 得到的流。那么流 f' 是一个最小代价流。

证明 通过图8-12说明证明过程。

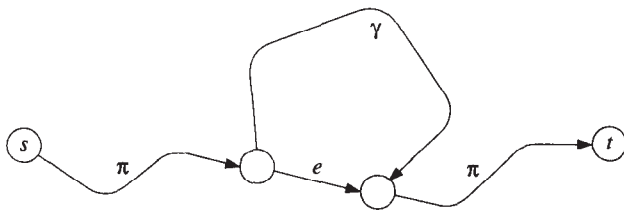


图8-12 定理8.7证明过程的说明

用反证法。假定 f' 没有最小代价。由定理8.5可知， f' 有一条负代价的增大回路 γ 。回路 γ 和路径 π 必定至少有一条公共边 e ，并且在与 π 的相反方向上遍历 e 。否则 γ 就会是关于流 f 的一条负代价的增大回路，这是不可能的，因为 f 有最小代价。考虑用 $\gamma - e$ 替换 π 中的边 e ，得到路径 $\hat{\pi}$ 。路径 $\hat{\pi}$ 是关于流 f 的一条增大路径。且路径 $\hat{\pi}$ 上的代价为

$$w(\hat{\pi}) = w(\pi) + w(\gamma) < w(\pi)$$

这与假设 π 是关于流 f 的一条具有最小代价的增大路径相矛盾。 ■

从初始为空的流开始，反复应用定理8.7（见图8-13），计算一个具有最小代价的最大流。给定当前流 f ，给剩余图 R_f 中的每条边赋权值如下（回忆8.2.2节中剩余图的定义）。对于原网络中从 u 到 v 方向的每条边 e ， R_f 中从 u 到 v 的边上的权值为 $w(u, v) = w(e)$ ，用 (u, v) 表示，而从 v 到 u 的边 (v, u) 上的权值为 $w(v, u) = -w(e)$ 。可以利用Bellman-Ford算法（见7.1.2节）计算 R_f 中一条最短路径。因为由定理8.5可知， R_f 中不含负代价回路。因此，我们得到一个伪多项式时间的算法（见5.3.3节），可在 $O(|f^*|nm)$ 时间内计算一个具有最小代价 f^* 的最大流。

400

上述算法的执行例子如图8-13所示。

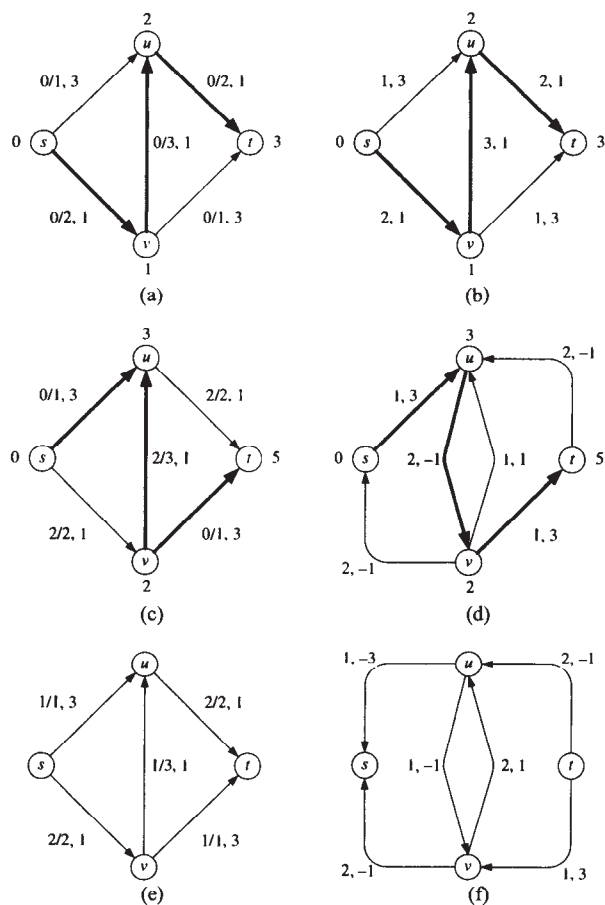


图8-13 通过连续最短路径增大方法计算最小代价流的例子。在每一步中，在左边显示网络，而在右边显示剩余网络。顶点上标记有它们离源点的距离。在网络中，每条边 e 标记有 $f(e)/c(e)$ ， $w(e)$ 。在剩余网络中，每条边标记有它的剩余容量和代价（剩余容量为0的边省略）。粗线表示增大路径。通过两次增大计算出最小代价流。在第一次增大中，沿着路径 (s, v, u, t) 推进两个单位的流。在第二次增加中，沿着路径 (s, u, v, t) 推进一个单位的流

401

8.4.3 修改权值

通过改变剩余图 R_f 中的权值使得它们全都为非负数来进行最短路径计算，可以减少所需的时间。修改之后，可以利用运行时间为 $O(m \log n)$ 的Dijkstra算法，而不是利用运行时间为 $O(nm)$ 的Bellman-Ford算法。

现在描述如何修改边上的权值。设 f 是当前最小代价流。用 $d_f(v)$ 表示顶点 v 距 R_f 中源点 s 的距离(distance)，定义为 R_f 中从 s 到 v 的一条路径上的最小权值（从源点 s 到顶点 v 的一条增大路径上的代价）。注意这个距离的定义不同于8.2.4节的Edmonds-Karp算法中使用的距离定义。

设 g 是从 v 沿着一条最小代价路径增大 f 所得到的流。定义 R_g 中边上权值 w' 的新集合如下（如图8-14所示）：

$$w'(u, v) = w(u, v) + d_f(u) - d_f(v)$$

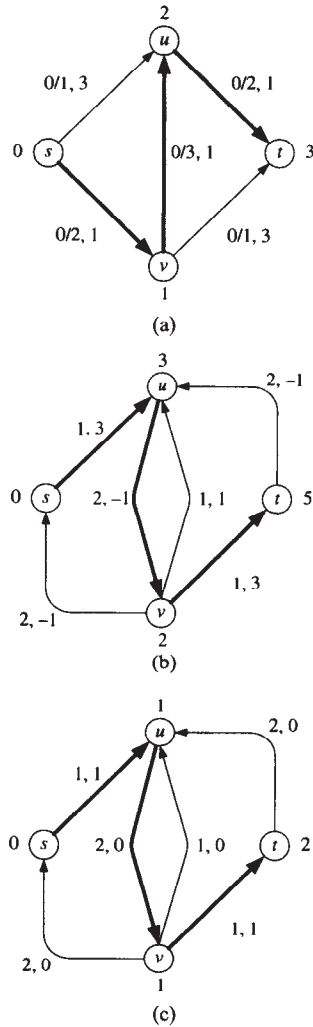


图8-14 利用连续最短路径增大方法,在计算最小代价流的过程中修改边上的代价。(a)流网络 N_f ,其初始流 f 为空,最短增大路径 $\pi_1 = (s, v, u, t)$,其代价 $w_1 = w(\pi_1) = 3$ 。每个顶点标记有距源点的距离 d_f (如图8-13b所示);(b)沿着路径 π 增大流 f 两个单位之后的剩余网络 R_g ,最短路径 $\pi_2 = (s, u, v, t)$,其代价 $w(\pi_2) = 5$ (同图8-13d);(c)修改过边上权值的剩余网络 R_g 。路径 π_2 仍然为一条最短路径。然而,它的代价减少了 w_1 。

引理8.8 对于剩余网络 R_g 中的每条边 (u, v) ,有

$$w'(u, v) \geq 0$$

同样,具有修改过边上权值 w' 的 R_g 中的一条最短路径也是具有原始边上权值 w 的一条最短路径。

证明 分两种情况考虑。

情况1: 边 (u, v) 在 R_f 中。

在这种情况下, v 与 s 的距离 $d_f(v)$ 不超过 u 与 s 的距离 $d_f(u)$ 加上边 (u, v) 的权值 $w(u, v)$,即

$$d_f(v) \leq d_f(u) + w(u, v)$$

由此可得

$$w'(u, v) \geq 0$$

情况2: 边 (u, v) 不在 R_f 中。

在这种情况下, (v, u) 必定是用于从流 f 得到流 g 的增大路径上的一条边, 且有

$$d_f(u) = d_f(v) + w(v, u)$$

因为 $w(v, u) = -w(u, v)$, 可得

$$w'(u, v) = 0$$

给定 R_g 中从 s 到 t 的一条路径 π , π 的关于修改过边权值的代价 $w'(\pi)$ 与 π 的代价 $c(\pi)$ 相差一个常数:

$$w'(\pi) = w(\pi) + d_f(s) - d_f(t) = w(\pi) - d_f(t)$$

因此, R_g 中关于原权值的一条最短路径也是一条关于修改过权值的最短路径。

利用连续最短路径方法计算最小代价流的完整算法由算法8-2 (MinCostFlow) 给出。

算法8-2 用于计算最小代价流的连续最短路径算法

算法 MinCostFlow(N):

输入: 加权流网络 $N = (G, c, w, s, t)$

输出: N 的具有最小代价 f 的最大流

for 每条边 $e \in N$ **do**

$f(e) \leftarrow 0$

for 每个顶点 $v \in N$ **do**

$d(v) \leftarrow 0$

$stop \leftarrow \text{false}$

repeat

计算加权剩余网络 R_f

for 每条边 $(u, v) \in R_f$ **do**

$w'(u, v) \leftarrow w(u, v) + d(u) - d(v)$

利用权值 w' 在 R_f 上运行Dijkstra算法

for 每个顶点 $v \in N$ **do**

$d(v) \leftarrow R_f$ 中的顶点 v 与 s 的距离

if $d(t) < +\infty$ **then**

{ π 是一条关于 f 的增大路径}

{计算 π 的剩余容量 $\Delta_f(\pi)$ }

$\Delta \leftarrow +\infty$

for 每条边 $e \in \pi$ **do**

if $\Delta_f(e) < \Delta$ **then**

$\Delta \leftarrow \Delta_f(e)$

{沿着路径 π 推进 $\Delta = \Delta_f(\pi)$ 个单位的流}

for 每条边 $e \in \pi$ **do**

if e 是一条前向边 **then**

$f(e) \leftarrow f(e) + \Delta$

else

$f(e) \leftarrow f(e) - \Delta$ { e 是一条后向边}

402
403

```

else
    stop ← true {f是具有最小代价的最大流}
until stop

```

以下定理概括了本节的内容。

404

定理8.8 对于具有 n 个顶点、 m 条边的网络，计算最小代价的最大流 f 所需时间为 $O(|f| m \log n)$ 。

8.5 Java 示例：最小代价流

在这一节里，提出了算法8-2（MinCostFlow）的一个Java实现。这个算法沿着最小代价的路径，利用连续增大方法计算一个最小代价流。基于模板方法模式，抽象类MinCostFlowTemplate实现了算法的核心功能。实现抽象类MinCostFlowTemplate任何具体类应该重写cost(e)和capacity(e)方法，返回特定应用中边上的代价值和容量值。代码段8-1中显示了MinCostFlowTemplate类中的实例变量及抽象方法cost(e)和capacity(e)。代码段8-2中显示了计算最短增大路径的核心方法doOneliteration()。

算法可以逐步运行或者一次运行完成。代码段8-3中所示的两个execute方法在给定的流网络上运行算法。算法将会运行，直至达到网络中给定的流的目标值，或者没有从源点（实例变量source_）到汇点（实例变量dest_）的更多增大路径。也可选择另一种方式，首先通过方法init(G, s, t)初始化算法（代码段8-4），然后反复调用doOneliteration()，一次一条增大路径地执行算法。代码段8-5中显示了Helper方法。可以调用方法cleanup()（未给出）删除算法中使用的所有辅助对象。

代码段8-6包含处理剩余图的方法。方法distance(v)返回顶点 v 距剩余图中源点的距离。方法residualWeight(e)返回边 e 上修改的权值。方法isAtCapacity(v, e)用于确定当从端点 v 遍历时，边 e 是否具有空的剩余容量，此外，调用方法doOneliteration()返回当前增大路径中边上的迭代器。这些结果由方法flow(e)和maximumFlow()（未给出）报告。

类MinCostFlowTemplate利用一个辅助类MinCostFlowDijkstra（未给出）使用Dijkstra算法计算最短路径。类MinCostFlowDijkstra是JDSL库[195]提供的泛型Dijkstra最短路径算法的特例（类似于代码段7-1~7-3给出的类）。MinCostFlowDijkstra中的主要方法如下。方法weight(e)返回边 e 上修改过的权值，它由类MinCostFlowTemplate中的方法residualWeight(e)计算得到。方法incidentEdges(v)也会被重写，以只考虑那些依附顶点 v 的非零剩余容量边（这个方法实际上会从剩余图中“删除”饱和边）。此外，通过利用修饰模式，MinCostFlowDijkstra记录当前最短路径上通过每个结点时的流瓶颈。因此，在MinCostFlowDijkstra每次执行结束时，就得到从源点到汇点的最小代价的路径，以及可以沿着这条路径推进的最大流量。

405

代码段8-1 类MinCostFlowTemplate中的实例变量和抽象方法

```

/**
 * Implementation of the minimum-cost flow algorithm based on
 * successive augmentations along minimum-cost paths. The algorithm
 * assumes that the graph has no negative-weight edges. The
 * implementation uses the template-method pattern.
 */
public abstract class MinCostFlowTemplate {

    // instance variables

```

```

protected MinCostFlowDijkstra dijkstra_;
protected InspectableGraph graph_;
protected Vertex source_;
protected Vertex dest_;
protected boolean finished_;
protected int maximumFlow_;
protected int targetFlow_;

// various constants
public final int ZERO = 0;
public final int INFINITY = Integer.MAX_VALUE;

// node decorations
private final Object FLOW      = new Object();
private final Object DISTANCE  = new Object();

/**
 * Returns the cost for a specified edge. Should be overridden for
 * each specific implementation.
 * @param e Edge whose cost we want
 * @return int cost for edge e (non-negative) */
protected abstract int cost(Edge e);

/**
 * Returns the capacity for the specified edge. Should be
 * overridden for each specific implementation.
 * @param e Edge whose capacity we want
 * @return int capacity for edge e (non-negative) */
protected abstract int capacity(Edge e);

```

406

代码段8-2 类MinCostFlowTemplate中的计算一条最小代价增大路径的方法

```

/**
 * Performs one iteration of the algorithm. The Edgelterator it
 * returns contains the edges that were part of the associated path
 * from the source to the dest.
 *
 * @return Edgelterator over the edges considered in the augmenting path */
public final Edgelterator doOneliteration()
throws jdsl.graph.api.InvalidEdgeException {
    Edgelterator returnVal;
    runDijkstraOnResidualNetwork();
    updateDistances();
    // check to see if an augmenting path exists
    if (distance(dest_) < INFINITY) {
        Edgelterator pathIter = dijkstra_.reportPath();
        int maxFlow = dijkstra_.reportPathFlow(dest_);
        maximumFlow_ += maxFlow;
        // push maxFlow along path now
        while (pathIter.hasNext()) {
            // check if it is a forward edge
            Edge e = pathIter.nextEdge();

            if (isForwardEdge(e)) {
                setFlow(e, flow(e) + maxFlow);
            } else {
                setFlow(e, flow(e) - maxFlow);
            }
        }
    }
    return returnVal;
}

```

```

    }
    pathIter.reset();
    returnVal = pathIter;
  } else {
    finished();
    returnVal = new EdgelteratorAdapter(new ArrayObjectIterator(new Object[0]));
  }
  return returnVal;
}

```

407

代码段8-3 类MinCostFlowTemplate中用于控制最小代价流算法执行的方法

```

/**
 * Helper method to continually execute iterations of the algorithm
 * until it is finished. */
protected final void runUntil() {
    while (shouldContinue()) {
        doOneIteration();
    }
}
/**
 * Execute the algorithm, which will compute the maximum flow
 * between source and dest in the Graph g.
 *
 * @param g a Graph
 * @param source of the flow
 * @param dest for the flow */
public final void execute(InspectableGraph g, Vertex source, Vertex dest)
    throws InvalidVertexException {
    init(g, source, dest);
    runUntil();
}
/**
 * Execute the algorithm, which will execute until the target flow
 * is reached, or no more flow is possible.
 *
 * @param g a Graph
 * @param source of the flow
 * @param dest for the flow */
public final void execute(InspectableGraph g, Vertex source, Vertex dest,
    int target)
    throws InvalidVertexException {
    targetFlow_ = target;
    execute(g, source, dest);
}

```

408

代码段8-4 类MinCostFlowTemplate中的初始化方法

```

/**
 * Initializes the algorithm. Set up all local instance variables
 * and initialize all of the default values for the decorations.

```

```

* Dijkstra's is also initialized.
*
* @param g a Graph
* @param source of the flow
* @param dest for the flow */
public void init(InspectableGraph g, Vertex source, Vertex dest)
    throws InvalidVertexException {
    if( !g.contains( source ) )
        throw new InvalidVertexException( source + " not contained in " + g );
    if( !g.contains( dest ) )
        throw new InvalidVertexException( dest + " not contained in " + g );
    graph_ = g;
    source_ = source;
    dest_ = dest;
    finished_ = false;
    maximumFlow_ = ZERO;
    targetFlow_ = INFINITY;
    // init dijkstra's
    dijkstra_ = new MinCostFlowDijkstra();
    dijkstra_.init(g, source);
    // initialize all the default values
    VertexIterator vertexIter = vertices();
    while (vertexIter.hasNext()) {
        Vertex u = vertexIter.nextVertex();
        setDistance(u, ZERO);
    }
    EdgelistIterator edgelistIter = edges();
    while (edgelistIter.hasNext()) {
        Edge e = edgelistIter.nextEdge();
        setFlow(e, ZERO);
    }
}

```

409

代码段8-5 类MinCostFlowTemplate中的Helper方法

```

/**
* Helper method to copy all of the vertex distances from an
* execution of Dijkstra's algorithm into local decorations so they
* can be used in computing the residual network for the next
* execution of Dijkstra's. */
protected void updateDistances() {
    // copy distances from residual network to our network
    VertexIterator vertexIter = vertices();
    while (vertexIter.hasNext()) {
        Vertex v = vertexIter.nextVertex();
        try {
            setDistance(v, dijkstra_.distance(v));
        } catch (InvalidQueryException iqe) {
            // vertex is unreachable: set distance to INFINITY
            setDistance(v, INFINITY);
        }
    }
}

```



```

}

/**
 * Helper method to execute Dijkstra's on the residual network. We
 * are sure to cleanup all past executions by first calling the
 * cleanup() method. */
protected void runDijkstraOnResidualNetwork() {
    dijkstra_.cleanup();
    dijkstra_.execute(graph_, source_, dest_);
}

/**
 * Helper method that is called exactly once when the algorithm is
 * finished executing. */
protected void finished() {
    finished_ = true;
}

```

410

代码段8-6 类MinCostFlowTemplate中的方法distance、residualWeight和isAtCapacity

```

/**
 * Returns the distance of a vertex from the source.
 * @param v a vertex
 * @return the distance of v from the source
 * @throws InvalidQueryException if v has not been reached yet */
public final int distance(Vertex v) throws InvalidQueryException {
    try {
        return ((Integer)v.get(DISTANCE)).intValue();
    }
    catch (InvalidAttributeException iae) {
        throw new InvalidQueryException(v+" has not been reached yet");
    }
}

/**
 * Returns the modified weight of edge e in the current residual
 * graph. It can be calculated on the fly because distance
 * information is only updated after every iteration of the
 * algorithm.
 * @param e Edge to find residual weight for
 * @return int residual weight of e */
public final int residualWeight(Edge e) {
    // use the absolute value because if we traverse
    // the edge backwards, then w(v,u) = -w(u,v)
    return Math.abs( cost(e) +
                     distance(graph_.origin(e)) -
                     distance(graph_.destination(e)) );
}

/**
 * Determines whether edge e has null residual capacity when
 * traversed starting at endpoint v.
 * @param v Vertex from which edge is being considered
 * @param e Edge to check

```

```

* @return boolean true if the edge is at capacity, false if not */
public final boolean isAtCapacity(Vertex v, Edge e) {
    // forward edges are full when capacity == flow
    if( v == graph_.origin( e ) )
        return (capacity(e) == flow(e));
    // back edges are full when flow == 0
    else
        return (flow(e) == 0);
}

```

411

8.6 习题

基础题

R-8.1 证明对于一个流 f ，流出源点的总量等于流入汇点的总量，即

$$\sum_{e \in E^+(s)} f(e) = \sum_{e \in E^-(t)} f(e)$$

R-8.2 对于图8-6a所示的流网络 N 和流 f ，回答以下问题：

- 找出增大路径 π 上的前向边和后向边。
- 有多少条关于流 f 的增大路径？对于每一条这样的路径，列出这条路径上的顶点序列和剩余容量。
- 计算 N 中的最大流的值。

R-8.3 利用引理8.4的证明中所用的方法，构造图8-4所示网络的一个最小割。

R-8.4 对于图8-2所示的流网络，说明Ford-Fulkerson算法的执行过程。

R-8.5 画出一个具有9个顶点和12条边的流网络。说明Ford-Fulkerson算法在其上的执行过程。

R-8.6 找出图8-7a所示流网络的一个最小割。

R-8.7 给定具有 m 条边的网络 N 中的一个最大流，证明可用 $O(m)$ 时间计算 N 的一个最小割。

R-8.8 找出图8-10a所示二分图的两个最大匹配，这些最大匹配不同于图8-10b中的最大匹配。

R-8.9 设 G 是一个完全二分图，对于每一对顶点 $x \in X$ ， $y \in Y$ ，满足 $|X| = |Y| = n$ ，那么存在一条连接 x 和 y 的边。证明 G 有 $n!$ 种不同的最大匹配。

R-8.10 对于图8-10b所示的流网络，说明Ford-Fulkerson算法的执行过程。

R-8.11 对于图8-7a所示的流网络，说明Edmonds-Karp算法的执行过程。

R-8.12 对于图8-2所示的流网络，说明Edmonds-Karp算法的执行过程。

R-8.13 证明引理8.7。

R-8.14 利用连续增大方法，沿着图8-13a所示流网络的负代价回路，说明最小代价流算法的执行过程。

R-8.15 利用连续增大方法，沿着图8-2所示流网络的最小代价路径，说明最小代价流算法的执行过程，其中边 (u, v) 的代价由 $|\deg(u) - \deg(v)|$ 给出。

412

R-8.16 算法8-2 (MinCostFlow) 是一个伪多项式时间的算法吗？

创新题

C-8.1 如果所有边上的容量限制为常数，那么Ford-Fulkerson算法最坏情况下的运行时间是多少？

C-8.2 改进引理8.6中的上界，证明Edmonds-Karp算法中至多存在 $nm/4$ 的增大量。

提示：除 $d_f(s, v)$ 之外，还利用 $d_f(u, t)$ 。

C-8.3 设 N 是具有 n 个顶点、 m 条边的流网络。证明如何用 $O((n+m)\log n)$ 时间计算具有最大剩余容量的一条增大路径。

C-8.4 证明如果在每次迭代中选择具有最大剩余容量的一条增大路径，则Ford-Fulkerson算法的运行时间为

$O(m^2 \log n \log |f^*|)$ 。

- C-8.5 你希望尽可能地增大一个网络的最大流，但只允许你增大一条边上的容量。
- 如何找到这样一条边（给出伪代码）？你可以假设存在计算最大流和最小割的算法。你所设计算法的运行时间是多少？
 - 总是可以找到这样的一条边吗？证明你的结论。
- C-8.6 给定一个流网络 N 和 N 的最大流 f ，假设 N 中边 e 的容量下降1，设 N' 是结果网络。通过修改 f ，给出一个计算网络 N' 的最大流的算法。
- C-8.7 给出一个确定具有 n 个顶点、 m 条边的图是否是二分图的 $O(n+m)$ 时间的算法。
- C-8.8 给出一个确定无向图的两个给定顶点 s 和 t 之间的边不相交路径最大数量的算法。
- C-8.9 计程车公司收到来自 n 个位置的请求。有 m 辆计程车可用，其中 $m \geq n$ ，计程车 i 距位置 j 的距离为 d_{ij} 。给出一个计算 n 辆计程车距 n 个位置的位移的算法，使总距离达到最短。
- C-8.10 给出计算最大流值的算法，满足以下另外两个约束条件：
- 每条边 e 上有一个通过流的下界 $\ell(e)$ 。
- 413 b. 有多个源点和汇点，并计算流的值作为所有源点流出的总流量（等于流入所有汇点的总流量）。
- C-8.11 在一个具有非整型容量的流网络中，证明Ford-Fulkerson算法可能不会终止。

程序设计

- P-8.1 设计和实现一个Java小程序，模拟Ford-Fulkerson流算法的执行过程。生动地表现流增大、剩余容量和实际流自身的变化过程。
- P-8.2 利用三种查找增大路径的不同方法实现Ford-Fulkerson流算法。并对这些方法仔细进行实验性比较。
- P-8.3 实现计算最大二分匹配的算法。表明如何重用计算最大流的算法。
- P-8.4 实现Edmonds-Karp算法。
- P-8.5 基于沿着负代价回路的连续增大方法，实现最小代价流算法。
- P-8.6 基于沿着最小代价路径的连续增大方法，实现最小代价流算法。利用Bellman-Ford算法实现该算法的一个变体，以及改变代价并利用Dijkstra算法实现该算法的一个变体。

8.7 本章注记

Ford和Fulkerson在其著作[71]中描述了网络流算法（8.2.2节）。Edmonds和Karp[65]描述了计算增大路径使得Ford-Fulkerson算法运行得更快的两种方法：最短增大路径（8.2.4节）和带有最大剩余容量的增大路径（习题C-8.4）。基于沿着最小代价路径的连续增大方法的最小代价流算法（8.4.2节）也是由Edmonds和Karp[65]提出的。

对图论算法和流网络感兴趣的读者，可进一步参阅Ahuja、Magnanti和Orlin[9]，Cormen、Leiserson和Rivest[55]、Even[68]、Gibbons[77]、Mehlhorn[149]、Tarjan[200]以及van Leeuwen[205]的著作。

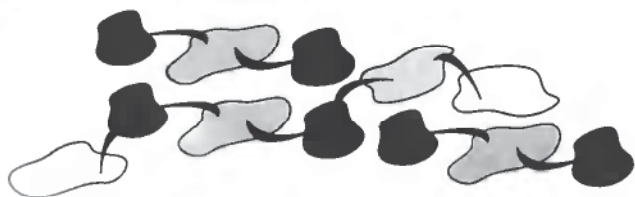
414 Dan Polivy开发了8.5节给出的最小代价流算法的实现。

Part 3

第三部分

因特网算法





文档处理迅速成为计算机中起着重要作用的功能之一。计算机可用于编辑文档、查找文档、在因特网上传输文档，以及在打印机和计算机显示器上显示文档。Web“冲浪”和Web查找涉及大量和重要的计算机应用问题，并且所有文本处理中的许多关键计算与字符串和字符串模式匹配有关。例如，因特网文档格式HTML和XML主要是文本格式，加上一些对多媒体内容的链接。理解因特网上的TB级信息需要进行大量的文本处理。

本章研究进行快速重要串操作的几个基本的文本处理算法。重点研究串查找和模式匹配算法，因为这些问题常常可能是许多文本处理应用中的计算瓶颈。还研究与文本处理有关的一些基本算法上的问题。

文本处理算法主要对字符串进行操作。本章中使用的串的术语和表示法相当直观，把一个串表示成为字符数组相当简单和有效。因而我们并不把大量注意力放在串表示上。然而，串的处理常常涉及模式匹配中的一个有趣的方法，因而在9.1节研究模式匹配算法。

9.2节研究trie数据结构，它是一种基于树的结构，可用于在串的集合中进行快速查找。

9.3节研究一个重要的文本处理问题，即文本文档的压缩问题，使得压缩后的文档存储更高效或者在网络上传输更高效。

最后9.4节研究的文本处理问题涉及如何度量两个文档之间的相似性问题。所有这些问题都是因特网计算中常常出现的主题，如Web爬虫、搜索引擎、文档分布和信息检索。

除了一些有趣的应用，本章的主题还强调某些重要的算法设计模式（见第5章）。尤其是在模式匹配这一节里，讨论蛮力方法（brute force method）。这种方法常常效率不高，但应用广泛。对于文本压缩，研究贪心法（5.1节）的应用，这种方法常常可以求出难解问题的近似解，且对于某些问题（如文本压缩应用问题），实际上可以产生问题的最优算法。最后在讨论文本相似性时，给出动态规划（5.3节）的另一种应用，对于某些初看起来要求指数时间的某些特例，应用这种方法解决问题的时间为多项式时间。

418

9.1 串和模式匹配算法

现代计算中文本文档无所不在，因为它们用于通信和发布信息。从算法设计的观点来看，可以把这样的文档看作简单的字符串，即可将构成文档的内容抽象成字符的序列。因此，对这样的数据进行查找操作和处理操作，就需要处理字符串的有效方法。

9.1.1 串操作

文本处理算法的核心是字符串的处理方法。字符串可以来自许多领域，包括科学、语言和因特网应用。以下给出这样一些串的例子。

```
P = "CGTAAACTGCTTTAATCAAACGC"
R = "U.S. Men Win Soccer World Cup!"
S = "http://www.wiley.com/college/goodrich/"
```

第一个串 P 来自DNA应用，最后一个串 S 是因特网地址（URL），表示本书的配套Web网址，中间的串 R 是一个虚构的新闻标题。本节提出一些用于处理这样一些串的串ADT支持的有用操作。

有几个典型的串处理操作涉及把一个大串分解成为若干较小的串。为了能够讨论由此操作导致的数据段，用子串这个术语表示 m 个字符的串 P 中形如 $P[i]P[i+1]P[i+2]\cdots P[j]$ 的串，其中 $0 \leq i \leq j \leq m-1$ ，即这个子串由 P 中下标从 i 到 j （含 i 和 j ）的字符构成。技术上讲，一个串实际上是自身的一个子串（取 $i=0$ 和 $j=m-1$ ）。因而，如果想要排除这种情况，必须把这个定义限制为真（proper）子串，这就要求 $i>0$ 或 $j<m-1$ 。为了简化子串表示，利用 $P[i..j]$ 表示 P 中下标从 i 到 j （含 i 和 j ）的子串，即

$$P[i..j] = P[i]P[i+1]\cdots P[j]$$

利用约定：如果 $i > j$ ，那么 $P[i..j]$ 等于长度为0的空串（null string）。此外，为了区分某些特殊形式的子串，称形如 $P[0..i]$ 的子串为 P 的前缀（prefix），其中 $0 \leq i \leq m-1$ ，称形如 $P[i..m-1]$ 的子串为 P 的后缀（suffix），其中 $0 \leq i \leq m-1$ 。例如，如果再次取 P 为上面给出的DNA串，那么"CGTAA"是 P 的前缀，"CGC"是 P 的后缀，"TTTAATC"是 P 的真子串。注意空串是任何其他串的前缀和后缀。

419

1. 模式匹配问题

在关于串的经典的模式匹配（pattern matching）问题中，给定一个长为 n 的文本（text）串 T 和一个长为 m 的模式（pattern）串 P ，想要查明 P 是否是 T 的一个子串。“匹配”的概念是存在 T 的一个从某个下标 i 开始的子串，它与 P 中字符逐个匹配，满足

$$T[i] = P[0], T[i+1] = P[1], \dots, T[i+m-1] = P[m-1]$$

即

$$P = T[i..i+m-1]$$

因此，一个模式匹配算法的输出要么是表明模式 P 不在 T 中的指示，要么是 T 中与 P 匹配的子串的起始下标。

为了清楚地表示字符串的一般概念，一般并不限制 T 和 P 中的字符明确来自于一个众所周知的字符集，如ASCII字符集或Unicode字符集。作为替代，通常利用常规符号 Σ 表示字符集，或者 T 和 P 中字符所属的字母表（alphabet）。这个字母表 Σ 当然可以是ASCII或Unicode字符集的一个子集，但字母表中的字符也可更多，甚至可以有无限个字符。然而，因为应用中使用的大多数文本处理算法所涉及的字符集是有限的，通常假设字母表 Σ 的大小为一个固定常数，用 $|\Sigma|$ 表示。

示例9.1 给定文本串

```
T = "abacaabaccabacabaabb"
```

和模式串

```
P = "abacab"
```

那么 P 是 T 的一个子串，即 $P = T[10..15]$ 。

在这一节里，介绍三种不同的模式匹配算法。

9.1.2 蛮力模式匹配

当需要查找某些内容或是优化某项功能时，蛮力算法设计方法是一种强有力的算法设计技术。在把这项技术应用于一般情形时，通常会枚举出涉及的所有可能的输入配置，并把所有这些枚举的配置中最好的挑出来。

蛮力模式匹配

在把这项技术应用于设计蛮力模式匹配算法（brute-force pattern algorithm）时，就可能导出所考虑的求解模式匹配问题的第一个算法——简单地测试 P 相对于 T 的所有可能的位置。算法9-1中所示的这种方法相当简单。

420

算法9-1 蛮力模式匹配

算法 BruteForceMatch(T, P):

输入：具有 n 个字符的串 T （文本）和具有 m 个字符的串 P （模式）

输出： T 中与 P 匹配的的第一个子串的起始下标，或者表明 P 不是 T 的子串的指示。

for $i \leftarrow 0$ to $n - m$ {对于 T 中的每个候选下标} do

$j \leftarrow 0$

 while ($j < m$ and $T[i + j] = P[j]$) do

$j \leftarrow j + 1$

 if $j = m$ then

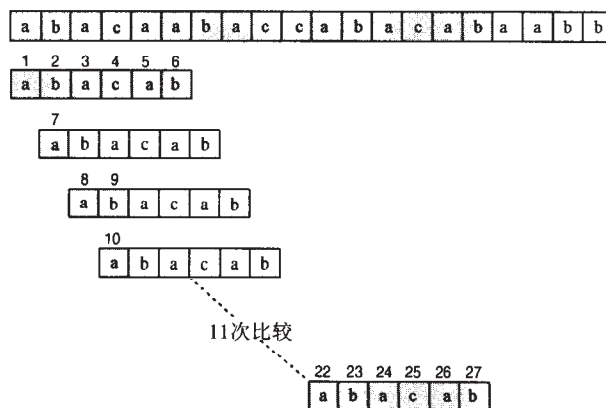
 return i

return " T 中不存在与 P 匹配的子串"

蛮力模式匹配算法不可能更简单。它由嵌套的两个循环组成，外层循环遍历文本中所有可能模式的起始下标，内层循环利用下标遍历模式中的每个字符，并把它与文本中潜在对应的字符进行比较。因此，直接可得蛮力模式匹配算法的正确性。

但是，蛮力模式匹配算法最坏情况下的运行时间不佳，这是由于对于 T 中的每个候选下标，可能要进行多达 m 个字符的比较，才能发现在当前下标上 P 与 T 不匹配。参照算法9-1，可见外层for循环至多执行 $n - m + 1$ 次，内层循环至多执行 m 次。因此，蛮力方法的运行时间为 $O((n - m + 1)m)$ ，即 $O(nm)$ 。注意，当 $m = n/2$ 时，这个算法的有一个二次运行时间 $O(n^2)$ 。

图9-1说明了蛮力模式匹配算法在串 T 和 P 上的执行过程，其中的串 T 和 P 来自示例9.1。



421

图9-1 蛮力模式匹配算法的运行示例。算法执行27次字符比较，图中标以数字以表示比较次数

9.1.3 Boyer-Moore 算法

起初，我们可能感觉总是需要检查 T 中的每个字符，以便定位作为子串的模式 P 。但并不总是这样，对于本节中研究的Boyer-Moore (BM) 模式匹配算法，有时可以避免对 P 和 T 中相当大一部分字符进行比较。唯一不同的是蛮力算法在字母表潜在地无限时仍能起作用，而BM算法假设字母表的大小固定、有限。当字母表为中等大小且模式相对较长时，它是最快的算法。

在这一节里，描述原始BM算法的一个简化版本。其主要思想是通过添加两个潜在地节省时间的启发式搜索，改进蛮力算法的运行时间：

照镜子的启发式搜索 (looking-glass heuristics)：当测试 P 相对于 T 的一种可能位移时，从 P 的末尾开始比较，并且向后移到 P 的前面。

字符跳跃的启发式搜索 (character-jump heuristics)：在测试 P 相对于 T 的一种可能位移时，当文本字符 $T[i] = c$ 与相应的模式字符 $P[j]$ 不匹配时，执行如下处理。如果 c 不包含在 P 中，那么使 P 的移动完全越过 $T[i]$ （因为它不匹配 P 中的任何字符）。否则，移动 P 直到 P 中出现一个字符 c 与 $T[i]$ 对准。

我们稍后只在直觉级简短地表述这些启发式搜索方法，这些方法集成为一体来解决问题。照镜子的启发式搜索建立另一个启发式搜索，可以避免将 P 与 T 中整个字符组进行比较。至少在这种情况下，通过向后搜索过程可以更快到达目的地，因为如果考虑将 P 与 T 中某个位置的字符进行比较，当遇见不匹配的情况时，那么利用字符跳跃的启发式搜索，使 P 相对于 T 进行大量移动，就可能避免许多不必要的比较。在测试 P 相对于 T 的可能位移时，如果能较早地利用字符跳跃的启发式搜索，就能得到较大的回报。

于是，我们定义如何能把字符跳跃的启发式搜索集成进串的模式匹配算法中。为了实现这种启发式搜索，定义函数 $\text{last}(c)$ ， c 取自字母表，并且如果在文本中找到与 c 相等的字符，它并不与模式匹配，函数 $\text{last}(c)$ 就要确定把模式 P 移动多远。特别定义函数 $\text{last}(c)$ 如下：

- 如果 c 在 P 中， $\text{last}(c)$ 定义为 c 在 P 中最后（最右端）出现的下标。否则，传统上定义函数 $\text{last}(c) = -1$ 。

如果字符可用作数组下标，那么可用查找表容易地实现 last 函数。把有效计算这个表的方法留作一个简单的习题 (R-9.6)。 last 函数给出了需要进行字符跳跃启发式搜索的所有信息。算法9-2显示了BM模式匹配方法。图9-2说明了跳跃步骤。

422

算法9-2 Boyer-Moore模式匹配算法

算法 $\text{BMMatch}(T, P)$:

输入：具有 n 个字符的串 T （文本）和具有 m 个字符的串 P （模式）

输出： T 中与 P 匹配的的第一个子串的起始下标，或者表明 P 不是 T 的一个子串的指示

计算函数 last

$i \leftarrow m - 1$

$j \leftarrow m - 1$

repeat

if $P[j] = T[i]$ **then**

if $j = 0$ **then**

return i {一次匹配}

else

$i \leftarrow i - 1$

$j \leftarrow j - 1$

```

else
     $i \leftarrow i + m - \min(j, 1 + \text{last}(T[i]))$     {跳跃步骤}
     $j \leftarrow m - 1$ 
until  $i > n - 1$ 
return "T中不存在与P匹配的子串"

```

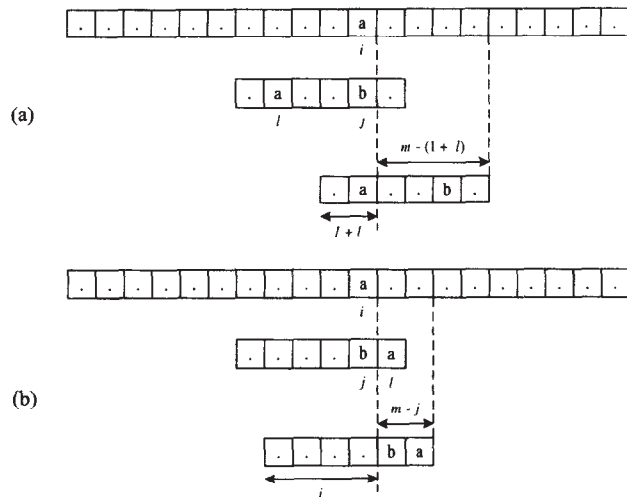
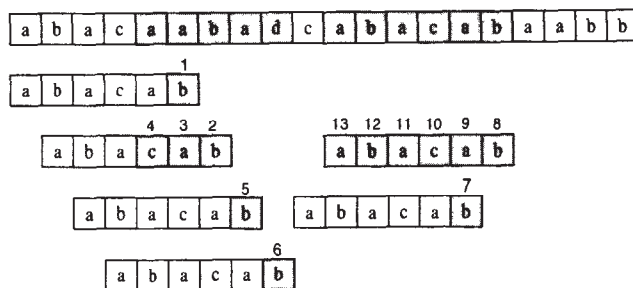


图9-2 BM算法中跳跃步骤的说明，其中 l 表示 $\text{last}(T[l])$ 。分两种情况：(a) $l + l \leq j$ ，其中把模式移动 $j - l$ 个单位；(b) $j < l + l$ ，其中把模式移动1个单位

423

对于类似于示例9.1中的输入串，图9-3说明了Boyer-Moore模式匹配算法在其上的执行过程。



last(c)函数				
c	a	b	c	d
last(c)	4	5	3	-1

图9-3 BM模式匹配算法的说明。算法进行13次字符比较，用数字标记指示出

BM模式匹配算法的正确性可由以下事实而得：每当方法执行一次移动时，都会保证不会跳过任何可能的匹配。因为 $\text{last}(c)$ 是 c 在 P 中最后出现的位置。

BM算法最坏情况下的运行时间为 $O(nm + |\Sigma|)$ ，即 last 函数计算所花的时间为 $O(m + |\Sigma|)$ ，实际查找模式最坏情况下所需的时间为 $O(nm)$ ，这个时间与蛮力算法的运行时间相同。达到最坏情况的一个文本-模式对的例子为

$$T = \begin{matrix} & 6 & 4 & 7 & 4 & 8 \\ a & a & a & a & a & a & \dots & a \end{matrix}$$

$$P = b \overbrace{a a \dots a}^{m-1}$$

然而，对于英语文本不太可能达到最坏情况的性能。

实际上，BM算法常常能够跳过很大一部分文本（如图9-4所示）。存在关于英语文本的实验证据，对于5个字符的模式串，对每个文本字符所做的平均比较次数约为0.24。但是，对于二进制串或者很短的模式，这个算法的回报不是太大，在这种情况下，9.1.4节讨论的KMP算法或者蛮力算法对于短模式的情况可能更好一些。

424

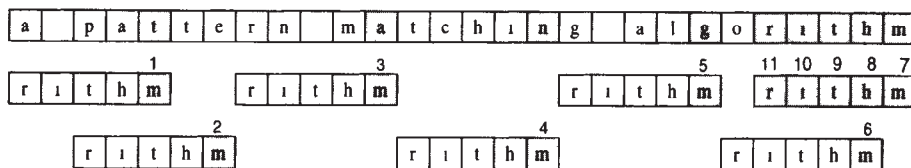


图9-4 Boyer-Moore算法对于英语文本和模式的执行过程，其中得到了显著的加速。注意，并不是所有文本字符都会被检查

实际上我们已经介绍了Boyer-Moore (BM) 算法的一种简化版本。通过把另一种移动启发式搜索过程应用到部分匹配的文本串上，原始BM算法的运行时间可以达到 $O(n + m + |\Sigma|)$ ，尽管这种方法对模式的移动要比字符跳跃的启发式搜索更大。这种备用的移动启发式搜索策略基于应用以下将要讨论的Knuth-Morris-Pratt模式匹配算法的主要思想。

9.1.4 Knuth-Morris-Pratt 算法

对于问题的特定实例，如示例9.1中给出的实例，在研究蛮力模式匹配算法和BM模式匹配算法最坏情况下的性能时，我们应该注意到大量无效的比较。确切地讲，在测试模式相对于文本的一个可能的位移时，可能进行许多次比较，然而如果找到一个模式字符与文本中的字符不匹配，那么就抛弃通过这些比较得到的所有信息，再次从模式的下一个增量位移从头开始进行比较。本节中讨论的Knuth-Morris-Pratt (或KMP) 算法避免了这个信息的浪费，利用这些信息，KMP算法最坏情况下可达到 $O(n + m)$ 的运行时间。它是最坏情况下的最优算法，即在最坏情况下，任何模式匹配算法都必定会把文本中的所有字符和模式中的所有字符至少检查一次。

1. 失效函数

KMP算法的主要思想是对模式串 P 进行预处理，以便计算失效函数 (failure function) f ，它表示对 P 所能进行的最大可能的正确移动，使得可以重用前面进行的比较结果。确切地讲，失效函数 $f(j)$ 定义为 P 的最长前缀的长度，且是 $P[1..j]$ 的一个后缀（注意我们并没有把 $P[0..j]$ 放在这里）。还利用了约定 $f(0) = 0$ 。后面将要讨论如何有效计算失效函数。失效函数的重要性在于它对模式自身内部的重叠子串进行“编码”。

425

示例9.2 考虑示例9.1中的模式串 $P = \text{"abacab"}$ 。对于串 P ，KMP的失效函数 $f(j)$ 如下表所示：

j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$f(j)$	0	0	1	0	1	2

算法9-3中所示的KMP模式匹配算法增量处理文本串 T ，并把它与模式串 P 进行比较。每次出现一个匹配之后，使当前下标增1。另一方面，如果发生不匹配的情况，且前面的过程已使 P 行进，那么通过失效函数确定需要继续相对于 T 检查 P 中的新下标。否则（发生不匹配的情况，且在 P 的开始处），只需增加 T 中的下标（并且使 P 的下标变量保持在它的开始处）。重复这个过程，直到找到 P 在 T 中的一个匹配，或者 T 的下标到达 T 的长度 n （表明在 T 中没找到模式 P ）。

算法9-3 KMP模式匹配算法

算法 KMPMatch(T, P):

输入: 具有 n 个字符的串 T （文本）和具有 m 个字符的串 P （模式）

输出: T 中与 P 匹配的第二个子串的起始下标，或者表明 P 不是 T 的一个子串的指示

$f \leftarrow \text{KMPSuccessFunction}(P)$ {构造 P 的失效函数 f }

$i \leftarrow 0$

$j \leftarrow 0$

while $i < n$ do

if $P[j] = T[i]$ then

if $j = m-1$ then

return $i - m + 1$ {一次匹配}

$i \leftarrow i + 1$

$j \leftarrow j + 1$

else if $j > 0$ {不匹配，但在 P 中已经向前推进} then

$j \leftarrow f(j-1)$ { j 的下标恰好在必须匹配的 P 的一个前缀之后}

else

$i \leftarrow i + 1$

return "T中不存在与P匹配的子串"

426

KMP算法的主要部分是while循环，它在每次迭代中，把 T 中的一个字符与 P 中的一个字符进行比较。取决于比较的结果，算法要么移动到 T 和 P 中的下一个字符，并通过失效函数获得 P 中新候选字符的位置，要么从 T 中的下一个下标开始。由失效函数的定义可得算法的正确性。所跳过的比较实际上是不需要的，因为失效函数保证所有忽略的比较是冗余的——它们会包括我们已经知道匹配的比较字符。

对于示例9.1中的相同输入串，图9-5说明了KMP模式匹配算法在这些串上的执行过程。注意失效函数的利用避免了重新比较模式串中的字符与文本串中的字符。另外注意，对于相同的串（图9-1），算法所做的总比较次数要少于蛮力算法所做的总比较次数。

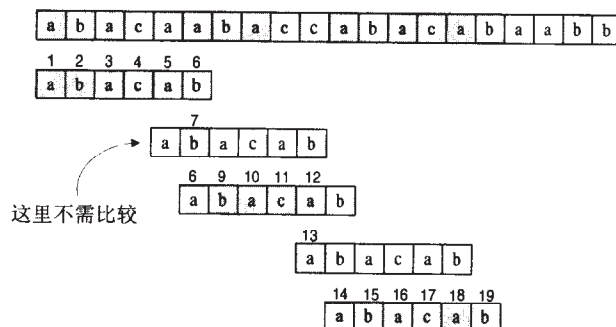


图9-5 KMP模式匹配算法的说明。这个模式的失效函数 f 在示例9.2中给出。算法进行19次字符比较，用数字标记指示

2. 性能

除了失效函数的计算之外，KMP算法的运行时间显然与while循环的迭代次数成正比。为了便于分析，定义 $k = i - j$ 。直观上讲， k 是模式 P 关于文本 T 所移动的总量。注意在算法的整个执行过程中，有 $k \leq n$ 。在循环的每次迭代中，将会出现以下三种情况之一。

- 如果 $\pi[i] = P[j]$ ，那么 i 增加1， k 不变，因为 j 也会增加1。
- 如果 $\pi[i] \neq P[j]$ 且 $j > 0$ ，那么 i 不变， k 至少增加1，因为在这种情况下， k 的变化范围为 $i - j \sim i - f(j-1)$ ，它会增加 $j - f(j-1)$ ，由于 $f(j-1) < j$ ，这个增量为正数。
- 如果 $\pi[i] \neq P[j]$ 且 $j = 0$ ，那么 i 增加1， k 也增加1，因为 j 不变。

因此，在循环的每次迭代中， i 或者 k 至少会增加1（可能两者都增加）；因而，KMP模式匹配算法中的while循环的迭代总次数至多为 $2n$ 。当然，要达到这个界限，假设已经计算出 P 的失效函数。

427

3. 构造KMP的失效函数

为了构造KMP模式匹配算法中使用的失效函数，利用算法9-4中显示的方法。这个算法是“自展法”的另一个例子，与KMPMatch算法中所用的过程相当类似。在KMP算法中，将模式与它自身做比较。每次遇见两个匹配的字符时，设 $f(i) = j + 1$ 。注意在算法的整个执行过程中，由于 $i > j$ ，当需要使用 $f(i)$ 时，它总是会被定义。

算法9-4 KMP模式匹配算法中使用的失效函数的计算。注意算法如何利用以前的失效函数值有效计算新的值

算法 KMPFailureFunction(P):

输入：具有 m 个字符的串 P （模式）

输出： P 的失效函数 f ，它把 j 映射到 P 中最长前缀（它是 $P[1..j]$ 的后缀）的长度。

$i \leftarrow 1$

$j \leftarrow 0$

$f(0) \leftarrow 0$

while $i < m$ do

 if $P[j] = P[i]$ then

 {已经匹配了 $j+1$ 个字符}

$f(i) \leftarrow j + 1$

$i \leftarrow i + 1$

$j \leftarrow j + 1$

 else if $j > 0$ then

 { j 的下标恰好在必须匹配的 P 的一个前缀之后}

$j \leftarrow f(j - 1)$

 else

 {这里不存在匹配}

$f(i) \leftarrow 0$

$i \leftarrow i + 1$

算法KMPFailureFunction的运行时间为 $O(m)$ 。它的分析类似于KMPMatch算法的分析。因此，可得：

定理9.1 Knuth-Morris-Pratt算法对于长为 n 的文本串和长为 m 的模式串进行模式匹配的运行时间为 $O(n + m)$ 。

KMP算法的运行时间分析初看上去令人有点惊讶，因为它声称其运行时间只与分别读入串 T 和 P 所需的时间成正比，就可以找到 T 中第一次出现的 P 。同时，也应注意KMP算法的运行时间不

428 依赖于字母表的大小。

9.2 trie

通过对模式进行预处理（在KMP算法中计算失效函数或者在BM算法中计算last函数）。上一节中介绍的模式匹配算法加快了文本查找的过程。在这一节里补充了一个方法，即给出预处理文本的串查找算法。这种方法适合于对固定文本进行一系列查询的应用，使得预处理文本的初始成本通过加快后续每次查询的速度得到补偿（例如，一个网址提供了莎士比亚的哈姆雷特的模式匹配，或者一个搜索引擎提供了关于哈姆雷特专题的Web页面）。

trie（发音为“try”）是一种基于树的数据结构，用于存储串以便支持快速模式匹配。trie主要应用在信息检索中。实际上，名字“trie”源于字“retrieval”。在信息检索应用中，如在基因数据库中查找某个DNA序列，给定串的集合 S ，所有定义利用同一字母表。

trie支持的主要查询操作是模式匹配和前缀匹配（prefix matching）。后一种操作涉及给定串 X ，查找 S 中以 X 为前缀的所有串。

9.2.1 标准 trie

令 S 是取自字母表 Σ 的 s 个串的集合，满足 S 中不存在一个串是另一个串的前缀。 S 的一个标准trie（standard trie）是一棵有序树 T ，满足如下性质（如图9-6所示）：

- 除了根之外， T 中的每个结点标记有 Σ 中的一个字符。
- T 中一个内部结点的子结点的次序由字母表 Σ 上的规范次序确定。
- T 有 s 个外部结点，每个外部结点关联 S 中的一个串，满足从根到 T 中一个外部结点 v 的路径上结点标记的连接产生 S 中关联 v 的一个串。

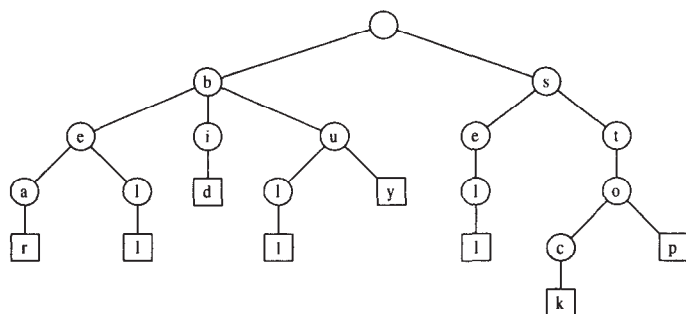


图9-6 串 {bear, bell, bid, bull, buy, sell, stock, stop} 的标准trie

因此，一个trie T 表示从根到 T 中外部结点的路径上 S 中的一个串。注意假设 S 中不存在一个串是另一个串的前缀。这保证 S 中的每个串唯一关联 T 中的一个外部结点。通过在每个串的末尾添加一个不在原始字母表 Σ 中的特殊字符，总是能够满足这个假设。

429 标准trie T 中的一个内部结点可拥有的子结点数介于 $1 \sim d$ 之间，其中 d 是字母表的大小。存在从根 r 到每个字符的根的子结点的一条边，该字符是集合 S 中某个串的第一个字符。此外，从 T 的根到深度为 i 的一个内部结点 v 的一条路径对应于 S 中串 X 的 i 个字符的前缀 $X[0..i-1]$ 。事实上，对于集合 S 中的串的前缀 $X[0..i-1]$ 后的每个字符 c ，存在 v 的标记有字符 c 的一个子结点。这样，一个trie简明地存储了存在于串集合之中的公共前缀。

如果字母表中只有两个字符，那么trie实际上是一棵二叉树，尽管某些内部结点可能只有一

个子结点（即可能是一棵不正确的二叉树）。一般而言，如果字母表中有 d 个字符，那么trie可能是一棵多路树，其中每个内部结点的子结点数介于 $1 \sim d$ 之间。例如，图9-6中显示的trie有几个内部结点只有一个子结点。可用一棵在其结点上存储字符的树实现一个trie。

以下定理提供了标准trie的一些重要的结构性性质：

定理9.2 存储总长为 n 、来自大小为 d 的字母表中 s 个串的集合 S 的标准trie具有如下性质：

- T 中每个内部结点至多有 d 个子结点。
- T 有 s 个外部结点。
- T 的高度等于 S 中最长串的长度。
- T 中的结点数为 $O(n)$ 。

性能

当两个串不共享公共非空前缀时，trie中的结点数出现最坏情况，即除了根之外，所有内部结点都有一个子结点。

串集合 S 的一个trie T 可用于实现其关键字是 S 中的串的字典，即通过在 T 中从根向下跟踪标记为 X 中字符的路径来查找串 X 。如果可以跟踪这条路径，并且终止于外部结点，那么可知 X 在字典中。例如，对于图9-6所示的trie，跟踪“bull”的路径终止于外部结点。如果路径不能被跟踪，或者路径可以被跟踪但终止于内部结点，那么 X 不在字典中。在图9-6的示例中，“bet”的路径不能被跟踪，而“be”的路径终止于一个内部结点。这两个单词都不在字典中。注意在这个字典的实现中，单个字符的比较代替了整个串（关键字）的比较。

容易看出查找大小为 m 的串的运行时间为 $O(dm)$ ，其中 d 为字母表的大小。实际上，至多访问 T 中的 $m+1$ 个结点，且每个结点上所需的时间为 $O(d)$ 。对于某些字母表，利用散列表或者查找表实现字符的字典，可以把每个结点上所需的时间改进到 $O(1)$ 或者 $O(\log d)$ 。然而，由于在大多数应用中 d 是常数，我们坚持认为简单的方法访问每个结点所需时间为 $O(d)$ 。

由上述讨论可知，可以利用trie实现一类特殊的模式匹配，称为字匹配（word matching），我们想要确定给定的模式是否与文本中的一个单词完全匹配（见图9-7）。字匹配不同于标准模式匹配，因为模式不能与文本中的任意子串匹配，只能与其中一个单词匹配。利用一个trie，长为 m 的字模式匹配所需时间为 $O(dm)$ ，其中 d 为字母表的大小，它与文本大小无关。如果字母表大小为常数（如自然语言和DNA串中的文本这种情况），查询所需时间为 $O(m)$ 。它与模式的大小成正比。对这种方法进行简单扩展，就支持前缀匹配查询。然而，这种方法对文本中出现的任意模式（例如，模式是一个字的真前缀或者跨两个字）不能进行有效的查询。

为了构造一个串集合 S 的标准trie，利用增量算法一次插入一个串。前面假设 S 中不存在一个串是另一个串的前缀。为了把串 X 插入当前trie T 中，首先试图跟踪 T 中关联 X 的路径。因为 X 尚未在 T 中，并且 S 中不存在一个串是另一个串的前缀，在到达 X 的末尾之前，将跟踪路径的过程停止在 T 的一个内部（internal）结点 v 上。然后创建 v 的结点后代的一个新链，存储 X 中的其余字符。插入 X 的时间为 $O(dm)$ ，其中 m 是 X 的长度， d 是字母表的大小。因此，构造集合 S 的整个trie所需时间为 $O(dn)$ ，其中 n 是 S 中串的总长度。

在标准trie中可能存在空间利用效率不高，这迅速推动了压缩trie（compressed trie）的开发，它也称为（由于历史原因）Patricia trie。也就是说，在标准trie中可能存在大量结点只有一个子结点，这样结点的存在是一种浪费，因为它蕴涵着树中的结点总数可能比相应文本中的字数要多。

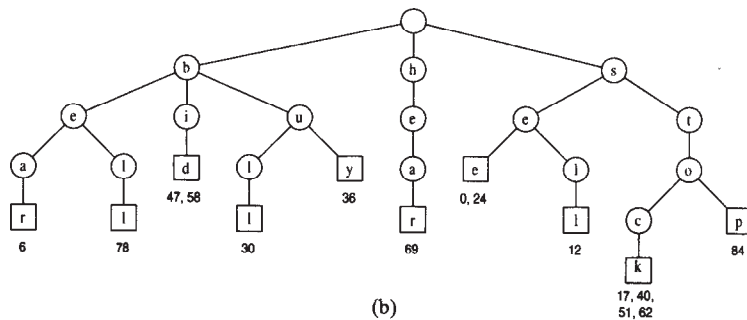
下一节将讨论压缩trie数据结构。

430

431

s	e	e		a		b	e	a	r	?		s	e	l	l		s	t	o	c	k	!	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
s	e	e		a		b	u	l	l	?		b	u	y		s	t	o	c	k	!		
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	
b	i	d		s	t	o	c	k	!		b	i	d		s	t	o	c	k	!			
47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68		
h	e	a	r		t	h	e		b	e	l	l	?		s	t	o	p	!				
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88				

(a)



(b)

432

图9-7 用一个标准trie进行字匹配和前缀匹配: (a)待查找的示例文本; (b)文本中字的标准trie (称冠词和介词为停止字 (stop word), 已去掉它们)。显示外部结点随着相应字的位置而增加

9.2.2 压缩 trie

压缩trie类似于标准trie, 但它能保证trie中的每个内部结点至少有两个子结点。通过把单子结点链压缩进各条边中来执行这个规则 (如图9-8所示)。设 T 是一棵标准trie。如果 T 的一个内部结点 v 有一个子结点且它不是根, 则称这个内部结点是冗余的 (redundant)。例如, 图9-6中的trie有8个冗余结点。如果

- v_i 是冗余的, 其中 $i = 1, \dots, k-1$
- v_0 和 v_k 不是冗余的

则称 $k \geq 2$ 条边的链

$$(v_0, v_1) (v_1, v_2) \cdots (v_{k-1}, v_k)$$

是冗余的。

通过把 T 中每条 $k \geq 2$ 条边的冗余链 $(v_0, v_1) (v_1, v_2) \cdots (v_{k-1}, v_k)$ 用单条边 (v_0, v_k) 代替, 可以把 T 转换成一个压缩trie, 并用结点 $v_1 \cdots v_k$ 的标记的连接重新标记 v_k 。

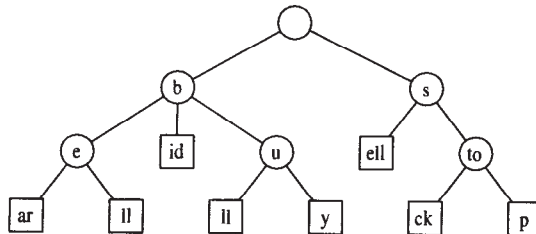


图9-8 串 {bear, bell, bid, bull, buy, sell, stock, stop} 的压缩trie。把它与图9-6中显示的标准trie进行比较

因此,用串标记压缩trie中的结点,这些串是集合中串的子串,而不是单个字符。压缩trie超过标准trie的优点在于,压缩trie中的结点数与串的个数成正比,而不是与串的总长度成正比,如下定理所示(与定理9.2进行比较)。

定理9.3 存储来自大小为 d 的字母表中 s 个串的集合 S 的压缩trie具有如下性质:

- T 中的每个内部结点至少有两个子结点,至多有 d 个子结点
- T 有 s 个外部结点
- T 中的结点数为 $O(s)$ 。

433

细心的读者可能想知道压缩路径是否提供了任何重要的优势,因为它被结点标记的相应扩展所抵消。实际上,仅当串的集合已经存储在主结构中,并且不需要实际存储集合中串的所有字符时,此时将压缩trie用作辅助(auxiliary)索引结构,压缩trie才真正有优势。但是,给定这个辅助结构,压缩trie的确相当有效。

例如,假定串的集合 S 是串 $S[0], S[1], \dots, S[s-1]$ 的一个数组。我们不是显式存储结点的标记 X ,而是用整数的三元组 (i, j, k) 隐式地表示它,满足 $X = S[i][j..k]$;即 X 是 $S[i]$ 的子串,由从 j 到 k 所包含的字符组成(见图9-9中的例子。同时与图9-7中的标准trie进行比较)。

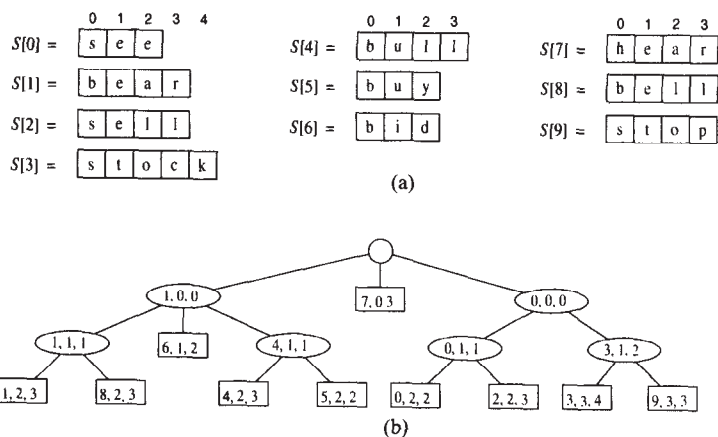


图9-9 (a)存储在数组中的串的集合 S ; (b) S 的压缩trie的压缩表示

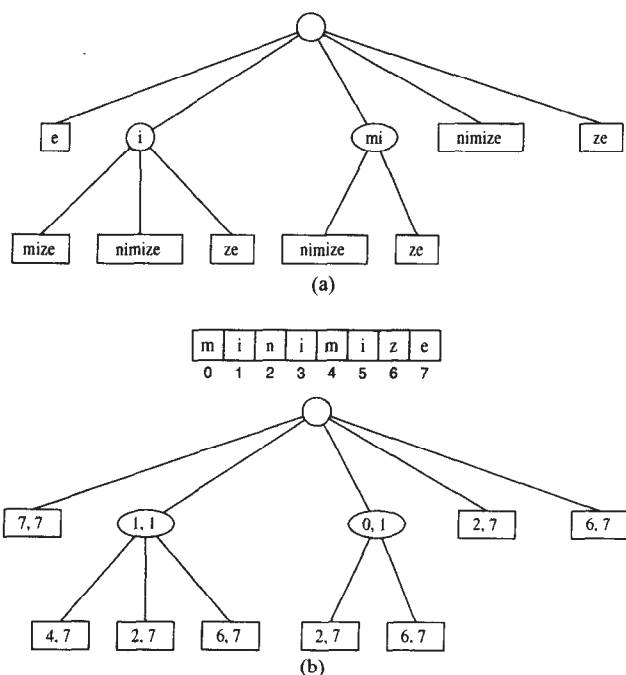
这种压缩模式使我们可以把trie的存储空间从标准trie的 $O(n)$ 降至压缩trie的 $O(s)$,其中 n 是 S 中串的总长度, s 是 S 中串的个数。当然,仍然必须存储 S 中的不同串,但减少了存储trie的空间。下一节介绍了一种应用,其中可以压缩存储串的集合。

434

9.2.3 后缀 trie

trie的主要应用之一出现在集合 S 中的串都是串 X 的后缀这种情况中,称这样的trie为串 X 后缀trie(suffix trie)(也称为后缀树或位置树)。例如,图9-10a中显示了串"minimize"的八个后缀的后缀树。

对于后缀trie,可进一步简化上一节中介绍的压缩表示,即可以构造trie,使得每个顶点的标记为数对 (i, j) ,表示串 $X[i..j]$ (见图9-10b)。为了满足不存在 X 的后缀是另一个后缀的前缀这个原则,可以在 X 的末尾添加一个特殊字符(因而每个后缀也会添加该字符),该特殊字符不在原始字母表 Σ 中,用 $\$$ 表示。也就是说,如果串 X 的长度为 n ,则可以构造 n 个串 $X[i..n-1]\$$ 的集合的一个trie $X[i..n-1]\$$,其中 $i = 0, \dots, n-1$ 。



435 图9-10 (a)串 $X = \text{"minimize"}$ 的后缀trie T ; (b) T 的压缩表示, 其中数对 (i, j) 表示串 $X[i..j]$

1. 节省空间

通过使用几种压缩技术, 利用后缀trie可以在标准trie上节省空间, 这些技术包括压缩trie中使用的那些技术。trie的压缩表示的优点在后缀trie上得到体现。因为长为 n 的串 X 的后缀总长度为

$$1 + 2 + \cdots + n = n(n+1)/2$$

显式存储 X 的所有后缀所需空间为 $O(n^2)$ 。即便如此, 后缀trie隐式地表示这些串所需空间为 $O(n)$, 正如下定理所阐述的那样。

定理9.4 长度为 n 的串 X 的后缀trie T 的压缩表示占用 $O(n)$ 的空间。

2. 构造

可用一个如9.2.1节中给出的增量算法, 构造长度为 n 的串的后缀trie。因为后缀的总长度是 n 的二次方, 因而构造过程所需时间为 $O(dn^2)$ 。然而, 用一个不同于构造一般trie的特殊算法, 可用 $O(n)$ 时间构造长度为 n 的串的(压缩)后缀trie。但是, 线性时间的构造算法相当复杂, 这里不做介绍。然而, 当想用后缀树解决其他问题时, 可以利用现有的快速构造算法。

3. 利用后缀trie

可用串 X 的后缀trie T 对文本 X 有效地执行模式匹配查询, 即通过跟踪 T 中关联 P 的一条路径, 可以确定一个模式 P 是否是 X 的一个子串。当且仅当可以跟踪到这条路径时, P 才是 X 的一个子串。算法9-5中给出了模式匹配算法的细节, 它假设在后缀trie的压缩表示中, 结点标记上具有如下另外一个性质:

如果结点 v 具有标记 (i, j) , Y 是关联从根到 v (已包含)的路径上长度为 y 的一个串, 那么 $X[y - y + 1..j] = Y$ 。

这个性质保证, 当出现一个匹配时, 可以容易地计算出模式在文本中的起始下标。

算法9-5 具有后缀trie的模式匹配。用 $(\text{start}(v), \text{end}(v))$ 表示结点 v 的标记，即指定关联 v 的文本子串的下标对

```

算法 suffixTrieMatch( $T, P$ ):
    输入: 文本 $X$ 的压缩后缀trie  $T$ 和模式 $P$ 
    输出:  $X$ 的一个子串与 $P$ 匹配时的起始下标, 或表明 $P$ 不是 $X$ 的一个子串的指示
     $p \leftarrow P.\text{length}()$  {要被匹配的模式后缀的长度}
     $j \leftarrow 0$  {要被匹配的模式后缀的起始下标}
     $v \leftarrow T.\text{root}()$ 
    repeat
         $f \leftarrow \text{true}$  {标志, 表明没有子结点被成功处理}
        for  $v$ 的每个子结点 $w$  do
             $i \leftarrow \text{start}(w)$ 
            if  $P[j] = T[i]$  then
                {处理子结点 $w$ }
                 $x \leftarrow \text{end}(w) - i + 1$ 
                if  $p \leq x$  then
                    {后缀比结点标记短, 或者与之具有相同的长度}
                    if  $P[j..j+p-1] = X[i..i+x-1]$  then
                        return  $i - j$  {匹配}
                    else
                        return " $P$ 不是 $X$ 的一个子串"
                else
                    {后缀比结点标记长}
                    if  $P[j..j+x-1] = X[i..i+x-1]$  then
                         $p \leftarrow p - x$  {更新后缀长度}
                         $j \leftarrow j + x$  {更新后缀起始下标}
                     $v \leftarrow w$ 
                     $f \leftarrow \text{false}$ 
                    break 跳出 for 循环
    until for  $T.\text{isExternal}(v)$ 
    return " $P$ 不是 $X$ 的一个子串"

```

437

4. 后缀trie的性质

算法suffixTrieMatch的正确性可由以下事实而得: 即向下查找trie T , 一次匹配模式 P 中的一个字符, 直到发生以下事件之一:

- 完全匹配模式 P ;
- 遇见一次不匹配 (由于for循环终止而引起的, 而该for循环不是经break语句终止的);
- 处理一个外部结点之后, P 中仍有要被匹配的字符。

设 m 是模式 P 的大小, d 是字母表的大小。为了确定算法suffixTrieMatch的运行时间, 执行以下观察:

- 至多处理trie中的 $m+1$ 个结点;
- 处理的每个结点至多有 d 个子结点;
- 在每个处理的结点 v 上, 对于 v 的每个子结点 w , 确定下次要处理 v 的哪个子结点至多需要进行一次字符比较 (利用快速字典索引 v 的子结点可能做出改进);
- 在处理的所有结点中, 至多进行总共 m 次字符比较;

- 每次字符比较所需时间为 $O(1)$ 。

5. 性能

由此可得, 算法suffixTrieMatch进行模式匹配查询的时间为 $O(dm)$ (如果利用字典索引后缀树中结点的子结点, 算法可能运行得更快)。注意运行时间并不依赖于文本 X 的大小。同时, 对于常数大小的字母表, 运行时间是模式大小的线性函数, 即为 $O(m)$ 。因此, 后缀trie适合于重复性模式匹配应用, 这种应用在固定文本上进行一系列模式匹配查询。

以下定理概括了本节的结论。

定理9.5 设 X 是一个取自大小为 d 的字母表中的 n 个字符组成的文本串。可用 $O(dm)$ 时间对 X 进行模式匹配查询, 其中 m 为模式长度, X 的后缀trie所用空间为 $O(n)$, 构造后缀trie的时间为 $O(dn)$ 。

438 下一节中探讨trie的另一种应用。

9.2.4 搜索引擎

WWW包含大量文本文档 (Web页面) 的集合。可以通过一个称为Web爬虫 (Web crawler) 的程序把这些这些页面中的信息收集起来, 然后把这些信息存储在一个特殊的字典数据库中。Web搜索引擎 (search engine) 使用户可以从这个数据库中检索相关信息, 从而把包含给定关键字的Web上的相关网页标识出来。在这一节里, 介绍搜索引擎的一个简化模型。

倒排文件

搜索引擎存储的核心信息是一个字典, 称为倒排索引 (inverted index) 或者倒排文件 (inverted file), 字典中存储关键字-值对 (w, L) , 其中 w 是一个字, L 是一个指向包含字 w 的页面的引用集合。称这个字典中的关键字 (字) 为索引项 (index item), 它应该是词汇表项和专有名词的一个尽可能大的集合。称这个字典中的元素为事件表 (occurrence list), 并且这些元素应该覆盖尽可能多的Web页面。

可以利用以下各项组成的数据结构, 有效实现一个倒排索引。

- 存储项的事件表的一个数组 (无特定次序);
- 索引项集的压缩trie, 其中每个外部结点存储相关项的事件表的索引。

把事件表存储在trie外部的原因是, 保持trie数据结构的规模足够小, 使之适合放置在内存中。另一方面, 由于事件表总规模较大, 它们必须存储在磁盘中。

有了数据结构, 对单个关键字的查询类似于字的匹配查询 (见9.2.1节), 即在trie中找到关键字并返回相关的事件表。

当给定多个关键字时, 希望的输出是包含所有 (all) 给定关键字的页面, 利用trie检索每个关键字的事件表, 并返回它们的交集。为了使求交集的计算简便, 每个事件表应该由按照地址排列的有序序列实现, 或者用字典实现 (例如, 见4.2节讨论的泛型归并计算)。

除了返回包含给定关键字的页面列表的基本任务之外, 搜索引擎还提供一项重要的服务, 按照相关性对返回的页面进行排序 (ranking)。对于计算机研究人员和电子商务公司而言, 为搜索引擎设计快速、精确的排序算法是一个重大的挑战。

439

9.3 文本压缩

在这一节里, 考虑另一个文本处理应用, 即文本压缩 (text compression)。在这个问题中, 给定一个定义在某个字母表 (如ASCII字符集或Unicode字符集) 上的串 X , 我们想要把 X 高效

编码为一个小的二进制串 Y （只利用字符0和1）。对于在一条低带宽的信道上进行通信的情况，如低速调制解调器线路或是无线连接，文本压缩是有用的，希望传输文本所需的时间最少。同样，文本压缩也用于更高效地存储大型文档集合，以便使固定容量的存储设备包含尽可能多的文档。

这一节探索的文本压缩方法是赫夫曼编码（Huffman code）。标准编码模式（如ASCII系统或Unicode系统）利用定长二进制串对字符编码（ASCII系统中码长为7位，Unicode系统中码长为16位）。另一方面，赫夫曼编码利用变长编码技术对串 X 进行优化。优化是基于字符出现的次数（frequency）进行的，对于每个字符 c ， $f(c)$ 表示 c 在串 X 中出现的次数。赫夫曼编码利用短码字串对出现次数多的字符进行编码，以及利用长码字串对出现次数少的字符进行编码，与定长编码相比，这样做节省了空间。

为了对串 X 进行编码，把 X 中的每个字符从它的定长码字转换为变长的码字，连接所有这些码字，产生 X 的一个编码 Y 。为了明确起见，在我们的编码中，确信不存在一个码字是另一个码字的前缀。称这样的码为前缀码（prefix code），它简化了 Y 的解码过程，以便恢复 X （见图9-11）。即使有这个限制，变长前缀编码所节省的空间也是巨大的，尤其是字符出现次数变化范围较大时（对于几乎每一种语言的自然语言文本也是如此）。

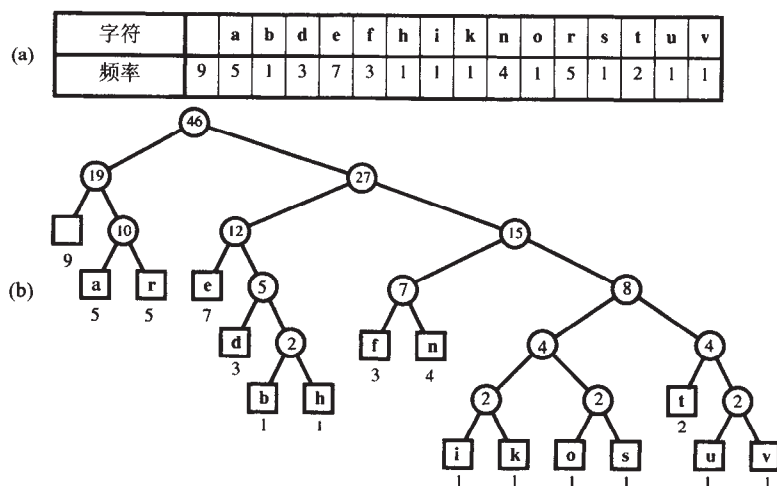


图9-11 对于输入串 $X = \text{"a fast runner need never be afraid of the dark"}$ 的赫夫曼编码示例的说明：

(a) X 中每个字符的次数；(b)串 X 的赫夫曼树 T 。通过跟踪从 T 的根到存储 c 的一个外部结点的路径，可得字符 c 的编码。左子结点表示0，右子结点表示1。例如，“a”的编码为010，“f”的编码为1100

用于产生 X 的最优变长前缀码的赫夫曼算法基于构造一棵表示编码的二叉树 T 。除了根之外， T 中的每个结点表示码字中的一位，其左子结点表示0，右子结点表示1。每个外部结点 v 关联一个特殊字符，那个字符的码字定义为从 T 的根到 v 的路径上关联结点的一个位序列（见图9-11）。每个外部结点 v 有一个次数（frequency） $f(v)$ ，简单地表示 X 中那个字符关联 v 的次数。此外， T 中每个内部结点 v 的次数 $f(v)$ 是以 v 为根的子树中所有外部结点次数之和。

440

9.3.1 赫夫曼编码算法

赫夫曼编码算法从串 X 的 d 个不同字符中的每个字符开始，将其编码成单结点二叉树的根结点。算法进行若干轮。在每一轮中，算法取两棵具有最小次数的二叉树，把它们合并成为一棵二

叉树。重复这个过程，直到只剩下一棵树（见算法9-6）。

算法9-6 赫夫曼编码算法

```

算法 Huffman( $X$ ):
    输入: 长为 $n$ 且有 $d$ 个不同字符的串 $X$ 
    输出:  $X$ 的编码树
    计算 $X$ 中每个字符 $c$ 出现的次数 $f(c)$ 
    初始化优先队列 $Q$ 
    for  $X$ 中的每个字符 $c$  do
        建立存储 $c$ 的单结点二叉树 $T$ 
        把树 $T$ 插入关键字为 $f(c)$ 的 $Q$ 中
    while  $Q.size() > 1$  do
         $f_1 \leftarrow Q.minKey()$ 
         $T_1 \leftarrow Q.removeMin()$ 
         $f_2 \leftarrow Q.minKey()$ 
         $T_2 \leftarrow Q.removeMin()$ 
        建立一棵新的二叉树 $T$ ，其左子树为 $T_1$ ，右子树为 $T_2$ 。
        把树 $T$ 插入关键字为 $f_1 + f_2$ 的 $Q$ 中
    return 树 $Q.removeMin()$ 

```

利用一个堆表示的优先队列，赫夫曼算法中while循环的每次迭代可用 $O(\log d)$ 时间实现。此外，每次迭代都会从 Q 中取出两个结点，并加入一个结点，这个过程重复 $d-1$ 次，此时 Q 中恰好只剩下一个结点。因此，这个算法的运行时间为 $O(n + d \log d)$ 。尽管充分证明这个算法的正确性超出了本书的范围，我们注意到它的直觉来自于一个简单的思想——任何最优编码可以转换成如下最优编码：在这个最优编码中，两个最少出现的字符 a 和 b 的码字只在它们的最后一位上不同。用一个字符 c 代替含有 a 和 b 的串，重复这个证明过程，得到如下定理：

定理9.6 对于长为 n 且有 d 个不同字符的串，赫夫曼算法用 $O(n + d \log d)$ 时间构造一个最优前缀编码。

9.3.2 修正贪心法

用于构建最优编码的赫夫曼算法是称为贪心法（greedy method）的算法设计方法的一个应用例子。回忆5.1节可知，这个设计模式可应用于优化问题，在这些优化问题中，我们试图构造某个结构，同时最小化或最大化那个结构的某个性质。

实际上，赫夫曼编码算法密切遵循贪心法模式的一般表述。即为了利用贪心法求解给定的优化代码问题，需要进行一系列选择。这个选择序列开始于一个得到良好理解的起始条件，并计算那个初始条件的成本。最终，通过从当前可能的所有选择中确定可以获得最佳成本改进的决策，来迭代执行额外的选择。这种方法并非总会得出一个最优解，但是当依据赫夫曼算法的方法使用它时，的确会找到最优前缀编码。

赫夫曼编码算法的全局最优性是基于如下事实：即最优前缀编码问题具有贪心选择（greedy choice）的性质。这个性质是从一个良好定义的初始条件开始，通过一系列局部最优选择达到全局最优（即在当前可用的可能性中做出最好的选择）。事实上，计算最优变长前缀编码正是具有贪心选择性质的问题的一个例子。

9.4 文本相似性测试

在遗传学和软件工程领域出现的一个普遍的文本处理问题是测试两个文本串之间的相似性。在遗传学应用中，两个串对应两条DNA链，例如，它们可能来自两个个体，如果它们对应的DNA序列有一条长的公共子序列，则认为它们在遗传上是相关的。同样，在软件工程应用中，两个串可能来自同一程序的源代码的两个版本，希望确定两个版本之间有哪些变化。此外，搜索引擎的数据收集系统（称为Web蜘蛛（spider）或爬虫（crawler））必定能够区分相似的Web页面，避免不必要的Web页面请求。实际上，确定两个串之间的相似性被看作是一个如此普遍的操作，以至于在Unix/Linux操作系统中提供了一个称为diff的程序，用于比较文本文件。

9.4.1 最长公共子序列问题

有几种不同的方式可以定义两个串之间的相似性。即便如此，还可以利用字符串及其子序列，抽象出这个问题的一个简单而通用的版本。给定一个大小为 n 的串 X ， X 的一个子序列（subsequence）是一个形如

$$X[i_1]X[i_2]\cdots X[i_k], \quad i_j < i_{j+1}, \quad \text{对于 } j = 1, \dots, k$$

的串，即它是一个不一定连续但按顺序取自 X 中的字符序列。例如，串AAAG是串CGATAATTGAGA的一个子序列。注意串的子序列的概念不同于9.1.1节中定义的串的子串的概念。

问题定义

这里阐述的文本相似性问题是**最长公共子序列（longest common subsequence, LCS）问题

。在这个问题中，给定两个取自某个字母表且大小分别为 n 、 m 的字符串 X 和 Y ，要求找出一个最长的串 S ，它同时是 X 和 Y 的一个子序列。

求解最长公共子序列问题的一种方法是枚举出 X 的所有子序列，并取其中同时也是 Y 的子序列的一个最大的子序列。由于 X 中的每个字符要么在子序列中，要么不在子序列中，因而可能有 2^n 种 X 的不同子序列。对于每一个子序列，需要 $O(m)$ 的时间确定它是否是 Y 的一个子序列。因此，蛮力方法产生运行时间为 $O(2^n m)$ 的一个指数算法，这个算法效率极低。在这一节里，讨论如何利用动态规划（dynamic programming，5.3节）快得多地求解这个最长公共子序列问题。

443

9.4.2 应用动态规划求解 LCS 问题

利用动态规划求解LCS问题，可以得到比指数时间快得多的算法。正如在5.3节中提到的那样，动态规划技术关键的步骤之一是定义满足子问题最优性质和子问题重叠性质的简单的子问题。

回忆在LCS问题中，给定两个长度分别为 n 和 m 的字符串 X 和 Y ，要求找出一个最长的串 S ，它同时是 X 和 Y 的一个子序列。因为 X 和 Y 是字符串，自然可以利用下标集定义子问题——串 X 和 Y 的下标。于是，我们定义子问题为计算 $X[0..i]$ 和 $Y[0..j]$ 的最长公共子序列长度的子问题，用 $L[i, j]$ 表示这个长度。

这个定义允许根据最优子问题解重写 $L[i, j]$ 。考虑如下两种情况（如图9-12所示）。

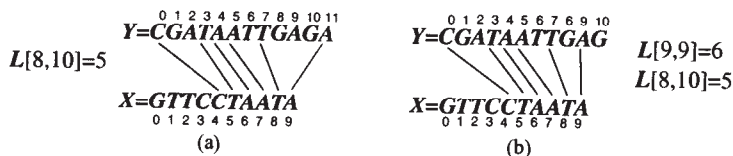


图9-12 定义 $L[i, j]$ 的两种情况：(a) $X[i] = Y[j]$ ；(b) $X[i] \neq Y[j]$

情况1: $X[i] = Y[j]$

令 $c = X[i] = Y[j]$ 。我们声明 $X[0..i]$ 和 $Y[0..j]$ 的最长公共子序列以 c 结束。为了证明这个声明, 假设它不为真。必定存在某个最长的公共子序列 $X[i_1]X[i_2]\cdots X[i_k] = Y[j_1]Y[j_2]\cdots Y[j_k]$ 。如果 $X[i_k] = c$ 或 $Y[j_k] = c$, 那么设置 $i_k = i$ 和 $j_k = j$, 可以得到同一序列。否则, 如果 $X[i_k] \neq c$, 那么把 c 添加到序列末尾, 得到一条更长的公共子序列。因此, $X[0..i]$ 和 $Y[0..j]$ 的最长公共子序列以 $c = X[i] = Y[j]$ 结尾。于是, 设

$$L[i, j] = L[i-1, j-1] + 1, \text{ 如果 } X[i] = Y[j] \quad (9.1)$$

情况2: $X[i] \neq Y[j]$

在这种情况下, 公共子序列中不能同时包含 $X[i]$ 和 $Y[j]$, 即公共子序列可能以 $X[i]$ 、 $Y[j]$ 结尾, 或者末尾不包含 $X[i]$ 或 $Y[j]$, 但不会同时包含 $X[i]$ 和 $Y[j]$ 。于是, 设

$$L[i, j] = \max\{L[i-1, j], L[i, j-1]\}, \text{ 如果 } X[i] \neq Y[j] \quad (9.2)$$

为了使公式(9.1)和公式(9.2)对于 $i = 0$ 或 $j = 0$ 时的边界情况有意义, 定义 $L[i, -1] = 0$ (其中 $i = -1, 0, 1, \dots, n-1$) 和 $L[-1, j] = 0$ (其中 $j = -1, 0, 1, \dots, m-1$)。

444

1. LCS算法

上述 $L[i, j]$ 的定义满足子问题的最优性, 因为如果没有子问题的最长公共子序列, 就不能得到问题的最长公共子序列。同样, 它利用了子问题的重叠性质, 因为子问题 $L[i, j]$ 的解可用于其他问题中 (即问题 $L[i+1, j]$ 、 $L[i, j+1]$ 和 $L[i+1, j+1]$)。

把 $L[i, j]$ 的定义转变成算法实际上相当直观。对于 $i = 0$ 或 $j = 0$ 时的边界情况, 初始化一个 $(n+1) \times (m+1)$ 的数组 L , 即初始化 $L[i, -1] = 0$ (其中 $i = -1, 0, 1, \dots, n-1$) 和 $L[-1, j] = 0$ (其中 $j = -1, 0, 1, \dots, m-1$) (这里存在表示上的轻微乱用, 因为在现实中, L 的行、列下标值从0开始)。然后, 反复构建 L 中的值, 直到计算出 X 和 Y 的最长公共子序列的长度 $L[n-1, m-1]$ 。算法9-7中给出这个方法所导致的最长公共子序列 (LCS) 问题的动态规划算法的伪代码描述。注意算法只存储了 $L[i, j]$ 的值, 而没有存储匹配。

算法9-7 LCS问题的动态规划算法

算法 LCS(X, Y):

输入: 给定两个元素个数分别为 n 和 m 的串 X 和 Y

输出: 对于 $i = 0, 1, \dots, n-1, j = 0, 1, \dots, m-1$, 输出 $X[0..i]$ 和 $Y[0..j]$ 的最长公共子序列的长度 $L[i, j]$ 。

for $i \leftarrow -1$ **to** $n-1$ **do**

$L[i, -1] \leftarrow 0$

for $j \leftarrow 0$ **to** $m-1$ **do**

$L[-1, j] \leftarrow 0$

for $i \leftarrow 0$ **to** $n-1$ **do**

for $j \leftarrow 0$ **to** $m-1$ **do**

if $X[i] = Y[j]$ **then**

$L[i, j] \leftarrow L[i-1, j-1] + 1$

else

$L[i, j] = \max\{L[i-1, j], L[i, j-1]\}$

return 数组 L

2. 性能

容易分析算法9-7的运行时间, 因为它由两个嵌套for循环支配, 外层的for循环迭代 n 次, 内

445

层的for循环迭代 m 次。因为循环内的每个if语句和赋值都需要 $O(1)$ 个基本操作。因而这个算法的运行时间为 $O(mn)$ 。因此, 动态规划技术可应用于最长公共子序列问题, 能够大大改进LCS问题的蛮力求解的指数时间。

算法LCS (9-7) 计算最长公共子序列的长度 (存储在 $L[n-1, m-1]$ 中), 但并没有计算最长公共子序列自身。正如在以下定理中所示的那样, 通过一个简单的后处理步骤, 从算法返回的数组 L 中可以提取最长公共子序列。

定理9.7 给定具有 n 个字符的串 X 和具有 m 个字符的串 Y , 可以用 $O(mn)$ 时间找到 X 和 Y 的最长公共子序列。

证明 观察得知算法LCS用 $O(mn)$ 时间计算输入串 X 和 Y 的最长公共子序列的长度。给定 $L[i, j]$ 值的表, 可以直观地构造一个最长公共子序列。一种方法是从 $L[n-1, m-1]$ 开始, 通过表执行回溯过程, 从后向前重构一个最长公共子序列。对于任何位置 $L[i, j]$, 确定是否 $X[i] = Y[j]$ 。如果是, 那么取 $X[i]$ 作为子序列的下一个字符 (注意 $X[i]$ 在找到的前一个字符 (如果有) 之前), 并向下移到 $L[i-1, j-1]$ 。如果 $X[i] \neq Y[j]$, 那么移到 $L[i, j-1]$ 和 $L[i-1, j]$ 之间较大者 (见图9-13)。当到达边界元素时 ($i = -1$ 或 $j = -1$), 算法停止。这个方法另外用 $O(m+n)$ 的时间构造一条最长公共子序列。 ■

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	4	4	4	4	4
6	0	1	1	2	2	3	3	3	4	4	5	5	5
7	0	1	1	2	2	3	4	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	5	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	6

0 1 2 3 4 5 6 7 8 9 10 11
 $Y = \text{CGATAATTGAGA}$

0 1 2 3 4 5 6 7 8 9
 $X = \text{GTTCTAATA}$

图9-13 从数组 L 构造一条最长公共子序列的算法说明

446

9.5 习题

基础题

- R-9.1 串 $P = \text{"aaabbbaaa"}$ 的非空前缀中, 有多少个也是 P 的后缀?
- R-9.2 画出一个图说明蛮力模式匹配算法对文本串 "aaabaadaabaaa" 和模式串 "aabaaa" 进行比较的过程。
- R-9.3 利用BM模式匹配算法重做上述问题, 计算last函数所执行的比较次数不计在内。
- R-9.4 利用KMP模式匹配算法重做上述问题, 计算失效函数所执行的比较次数不计在内。
- R-9.5 对于以下模式串, 计算BM模式匹配算法中使用的表示last函数的一个表

"the quick brown fox jumped over a lazy cat"

假设如下的字母表 (从空格字符开始):

$\Sigma = \{ , a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z \}$

- R-9.6 假设字母表 Σ 中的字符可被枚举, 且可以索引数组。给出一个从长为 m 的模式串 P 构造last函数的 $O(m + |\Sigma|)$ 时间的方法。

R-9.7 对于模式串"cgtaggttcgtac", 计算表示KMP失效函数的一个表。

R-9.8 对于以下串的集合, 画出一个标准trie:

{abab, baba, ccccc, bbaaaa, caa, bbaacc, cbcc, cbca}

R-9.9 画出习题R-9.8中给出的串的集合的一个压缩trie。

R-9.10 对于以下串, 画出一个后缀trie的压缩表示:

"minimize minime"

R-9.11 串"cgtaggttcgtac"的最长前缀是什么? 同时它还是这个串的一个后缀。

R-9.12 画出以下串的次数组和赫夫曼树:

"dogs do not spot hot pots or cats"

447 R-9.13 证明如何利用动态规划方法计算两个串"bababab"和"bbabbaaab"之间的最长公共子序列。

R-9.14 给出两个串

$X = \text{"skullandbones"}$

$Y = \text{"lullabybabies"}$

的最长公共子序列表 L 。这些串之间的最长公共子序列是什么?

创新题

C-9.1 给出长为 n 的文本 T 和长为 m 的模式 P 的例子, 使得蛮力模式匹配算法的运行时间为 $\Omega(nm)$ 。

C-9.2 对于长为 m 的模式, 给出KMPFailureFunction方法(算法9-4)运行时间为 $O(m)$ 的一个证明。

C-9.3 证明如何修改KMP串模式匹配算法, 使其找出作为 T 中子串出现的模式串 P 在 T 中的每次出现, 同时运行时间仍为 $O(n+m)$ (确信捕获甚至那些重叠的匹配)。

C-9.4 设 T 是长为 n 的文本, P 是长为 m 的模式。描述一个 $O(n+m)$ 时间的方法, 找出 P 的一个最长前缀, 它也是 P 的一个后缀。

C-9.5 如果存在下标 $0 \leq i \leq m$, 满足 $P = T[n-m+i..n-1] + T[0..i-1]$, 则称长为 m 的模式 P 是长为 n 的文本 T 的一个循环子串(circular substring)。也就是说, 如果 P 是 T 的一个子串或 P 等于 T 的一个后缀和 T 的一个前缀的连接。给出一个 $O(n+m)$ 时间的算法, 确定 P 是否是 T 的一个循环子串。

C-9.6 通过重新定义失效函数如下, 可以修改KMP模式匹配算法, 使它在二进制串上运行得更快:

$f(j) = \text{小于} j \text{的最大的} k, \text{满足} P[0..k-2] \overline{P[k-1]} \text{是} P[1..j] \text{的一个后缀}$

其中上划线 $\overline{P[k]}$ 表示 P 中第 k 位的补码。描述如何修改KMP算法, 使它能够利用这个新的失效函数, 并给出计算这个失效函数的方法。证明这个方法在文本和模式之间至多进行 n 次比较(并与9.1.4节中给出的标准KMP算法的 $2n$ 次比较进行对比)。

C-9.7 利用KMP算法的思想, 修改本章介绍的简化BM算法, 使其运行时间为 $O(n+m)$ 。

C-9.8 证明如何利用后缀trie进行前缀匹配查询。

C-9.9 给出从标准trie中删除一个串的有效算法, 并分析其运行时间。

C-9.10 给出从压缩trie中删除一个串的有效算法, 并分析其运行时间。

C-9.11 描述构造后缀trie的压缩表示的算法, 并分析其运行时间。

448 C-9.12 设 T 是长为 n 的文本串。描述一个找出 T 的最长前缀的 $O(n)$ 时间的方法, 该前缀也是 T 的逆串的一个子串。

C-9.13 描述一个找出最长回文的有效算法, 该回文是长为 n 的串 T 的一个后缀。回文(palindrome)是一个等于它的逆串的串。你的方法的运行时间是多少?

C-9.14 给定一个数的序列 $S = (x_0, x_1, x_2, \dots, x_{n-1})$, 描述一个找出数的最长子序列 $T = (x_{i_0}, x_{i_1}, x_{i_2}, \dots, x_{i_{k-1}})$ 的 $O(n^2)$ 时间的算法, 满足 $i_j < i_{j+1}$ 和 $x_{i_j} > x_{i_{j+1}}$, 即 T 是 S 的一个最长递减子序列。

C-9.15 描述上述问题的一个 $O(n \log n)$ 时间的算法。

C-9.16 证明对于 n 个不同数组成的序列, 存在一个大小至少为 $\lfloor \sqrt{n} \rfloor$ 的递减或递增序列。

C-9.17 把长度分别为 n 和 m 的两个串 X 和 Y 的编辑距离(edit distance)定义为把 X 变成 Y 所进行的编辑次数。一次编辑由一次字符插入、一次字符删除或者一次字符替换组成。串"algorithm"和串"rhythm"的编辑距

离为6。设计一个计算 X 和 Y 之间编辑距离的 $O(mn)$ 时间的算法。

C-9.18 设 A 、 B 和 C 是三个长为 n 的字符串，它们取自同一常数大小的字母表 Σ 。设计一个找出所有三个串 A 、 B 和 C 的一个最长公共子串的 $O(n^3)$ 时间的算法。

C-9.19 考虑当要求确定长为 m 的模式 P 是否包含在长为 n 的文本 T 中时，模式匹配算法总是返回“no”，这两个串均取自大小为 d 的同一字母表，其中常数 $d > 1$ 。这个简单的算法不正确地确定 P 是否是 T 的一个子串，给出一种表征这种可能性的大 O 表示，假设长为 m 的所有可能的模式串具有相等的概率。你的算法界限必定为 $o(1)$ 。

提示：当 m 很大时，这显然是一个准确的算法。

C-9.20 假定 A 、 B 和 C 是三个整型数组，表示三个大小都为 n 的字符串的ASCII值或Unicode值。给定任一整数 x ，设计一个 $O(n^2 \log n)$ 时间的算法，确定是否存在数 $a \in A$ ， $b \in B$ 和 $c \in C$ ，满足 $x = a + b + c$ 。

C-9.21 给出上一个问题的一个 $O(n^2)$ 时间的算法。

C-9.22 假定常数大小的字母表 Σ 中的每个字符 c 都有一个整数值 $w(c)$ 。两位选手准备进行一场比赛，给定取自 Σ 上的长为 $2n$ 的一个串 P 。在每一轮中，必须选定一位选手并删除 P 中的第一个或者最后一个字符，使其长度减1。每位选手的目标是最大化他/她所选择的所有字符的总价值。给出计算第一位选手的最优策略的 $O(n^2)$ 时间的算法。

提示：利用动态规划。

C-9.23 对于上一个习题中的比赛，为第一位选手设计一个 $O(n)$ 时间的非损失策略。你的策略不必是最优的，但应该保证结束时为平局或者结果对于第一位选手更好。

449

程序设计

P-9.1 利用在因特网上找到的文档，对变长模式的至少两种不同的模式匹配算法的效率进行一个实验性分析。

P-9.2 实现基于赫夫曼编码的压缩模式和解压缩模式。

P-9.3 建立用于实现ASCII串集合的标准trie和压缩trie的类。每个类应该有一个构造函数，以串的列表作为参数，类中应该包含一个方法，用于测试给定的串是否存储在trie中。

P-9.4 对于小型Web站点的页面，实现9.2.4节中描述的简化搜索引擎。利用站点上的页面中的所有字作为索引项，排除诸如冠词、介词和代词之类的停止字。

P-9.5 通过添加一个页面-排列特性到9.2.4节描述的简化搜索引擎中，实现一个小型Web站点页面的搜索引擎。你的页面-排列特性应该首先返回最相关的页面。利用页面中的所有字作为索引项，排除诸如冠词、介词和代词之类的停止字。

P-9.6 设计和实现求解最长公共子序列（LCS）问题的动态规划法和贪心法。运行实验，比较这两种方法的运行时间与它们所产生解的质量。

P-9.7 实现可以取任何文本串并产生其赫夫曼编码的算法。

9.6 本章注记

Knuth、Morris和Pratt在其期刊文章[122]中描述了KMP算法。Boyer和Moore同年[36]发表了他们的算法。在他们的文章中，Knuth等人[122]还证明了BM算法具有线性运行时间。最近，Cole[49]证明BM算法最坏情况下至多进行 $3n$ 次字符比较，这个界限是紧密的。本章介绍的所有算法也在Aho[6]的著作中讨论过，虽然是从更理论性的框架上进行讨论的。对串的模式匹配算法感兴趣的读者，可进一步参阅Stephen[193]、Aho[6]以及Crochemore和Lecroq[56]的著作。

Morrison[156]发明了trie一词，在Knuth[119]的经典著作Sorting and Searching中对其进行了广泛讨论。名称“Patricia”是“Practical Algorithm to Retrieve Information Coded in Alphanumeric”[156]的简写。McCreight[139]证明了如何用线性时间构造后缀trie。Baeza-Yates和Ribeiro-Neto[21]提供的信息检索领域的导论性著作包括Web搜索引擎的讨论。给出的编码问题的贪心法的应用来自Huffman[104]。

450



今天计算机用于大量敏感应用中。顾客通过电子商务购买商品和支付账单。企业利用因特网共享敏感的公司文档，并与商业伙伴进行交流。大学利用计算机网络存储学生的个人信息及其分数。如果这样的敏感信息被篡改、毁坏或者落入坏人之手，就可能遭到损坏。在这一章里，讨论几种强大的保护敏感信息的算法技术，以便达到如下目标：

- **数据完整性 (data integrity)**：不能不加检测地修改信息。例如，防止修改订单或其他以电子方式传输的合约性文档是重要的。
- **鉴别 (authentication)**：个人和组织对敏感信息的访问或通信必须经过正确确认即鉴别。例如，公司为其雇员提供的远程通信安排应该建立一种授权过程，以通过因特访问公司的数据库。
- **授权 (authorization)**：进行涉及敏感信息计算的代理必须被授权才能进行那些计算。
- **不可抵赖性 (nonrepudiation)**：交易隐含着一个合约，同意此合约的各方绝不能具有虽没被发觉而可逃避他们应尽义务的能力。
- **私密性 (confidentiality)**：敏感信息应该保持机密，未经授权的个人看不到那个信息，即必须保证发送者和接收者看到数据，而不能在通信时让未经授权的一方窃听到。例如，许多电子邮件消息是私密的。

本章中讨论的许多技术在达到上述目标时利用了数论知识。因此，本章首先讨论了许多重要的数论概念和算法。描述了一个计算最大公因子的古老但非常有效的欧几里得算法，以及计算模指数和逆元的算法。此外，由于素数在密码计算中起着非常关键的作用，还讨论一些测试一个数是否是素数的有效方法。我们表明有多少数论算法可用于实现计算机安全服务的密码算法中。重点关注了加密算法和数字签名，包括流行的RSA模式。本章还讨论了可以通过这些算法构建的几个协议。

最后讨论快速傅里叶 (FFT) 变换，它是一种常规的分治技术，这项技术可以解决许多具有类似乘法特性的问题。我们证明如何利用FFT有效解决多项式相乘问题和大整数相乘问题。还给出了大整数相乘的FFT算法的Java实现，并从实验角度比较了这种算法与标准大整数相乘算法的性能。

452

10.1 与数有关的基本算法

在这一节里，讨论了进行与数有关的重要计算的几种基本算法。描述了计算模 n 指数、模 n

乘法逆元和测试一个整数 n 是否是素数的有效方法。所有这些计算都有若干重要应用，包括构成著名的密码计算中的核心算法。在可以介绍这些算法之前，首先必须给出数论的一些基本事实。在讨论过程中，假设所有变量都是整数。同样，把某些数学事实的证明留作习题。

10.1.1 基本数论的一些事实

为了进入主题，需要一些基本数论的事实，包括某些符号和定义。给定正整数 a 和 b ，利用符号

$$a|b$$

表明 a 整除 (divide) b ，即 b 是 a 的倍数。如果 $a|b$ ，那么可知存在整数 k ，满足 $b = ak$ 。由此定义直接可得整除性的以下性质：

定理10.1 设 a 、 b 和 c 是任意整数。那么

- 如果 $a|b$ 且 $b|c$ ，那么 $a|c$ 。
- 如果 $a|b$ 且 $a|c$ ，那么对于所有整数 i 和 j ，都有 $a|(ib + jc)$ 。
- 如果 $a|b$ 且 $b|a$ ，那么 $a = b$ 或者 $a = -b$ 。

证明 见习题R-10.1。 ■

如果整数 $p \geq 2$ 且1和 p 是它的唯一因子，则称 p 是一个素数 (prime)。因此，在 p 是素数的情况下， $d|p$ 蕴涵着 $d = 1$ 或者 $d = p$ 。称大于2且不是素数的整数为合数 (composite)。因此，例如：5、11、101和98 711是素数，而25和10 403 ($=101 \cdot 103$)是合数。同时可得以下定理：

定理10.2 (基本运算定理) 设 $n > 1$ 是一个整数。那么存在唯一素数集合 $\{p_1, \dots, p_k\}$ ，以及正整数幂 $\{e_1, \dots, e_k\}$ ，满足

$$n = p_1^{e_1} \dots p_k^{e_k}$$

在这种情况下，称乘积 $p_1^{e_1} \dots p_k^{e_k}$ 为 n 的素数分解 (prime decomposition)。定理10.2和素数分解的唯一性概念是几种密码模式的基础。

453

1. 最大公因子 (GCD)

正整数 a 和 b 的最大公因子 (greatest common divisor) 是整除 a 和 b 的最大整数，用 $\gcd(a, b)$ 表示。另外，也可称 $\gcd(a, b)$ 为数 c ，满足如果 $d|a$ 且 $d|b$ ，那么 $d|c$ 。如果 $\gcd(a, b) = 1$ ，称 a 和 b 是互素的 (relatively prime)。利用如下两个规则，可把最大公因子的概念扩展到任意整数对：

- $\gcd(a, 0) = \gcd(0, a) = a$ 。
- $\gcd(a, b) = \gcd(|a|, |b|)$ ，它可以处理负值。

因此， $\gcd(12, 0) = 12$ ， $\gcd(10403, 303) = 101$ ， $\gcd(-12, 78) = 6$ 。

2. 模运算符

按序介绍几个关于模运算符 (modulo operator) (mod) 的词。回忆 $a \bmod n$ 是 a 被 n 除所得的余数，即

$$r = a \bmod n$$

意思是

$$r = a - \lfloor a/n \rfloor n$$

换句话说，存在某个整数 q ，满足

$$a = qn + r$$

此外, 注意 $a \bmod n$ 总是集合 $\{0, 1, 2, \dots, n-1\}$ 中的一个整数, 即使 a 为负数时也是如此。

有时讨论模 n 同余 (congruence) 更方便, 如果

$$a \bmod n = b \bmod n$$

则称 a 与 b 是模 n 同余的, 称之为模数 (modulus), 可写作

$$a \equiv b \pmod{n}$$

于是, 如果 $a \equiv b \pmod{n}$, 那么对于某个整数 k , 有 $a - b = kn$ 。

3. 模运算符和GCD的关系

以下定理给出了最大公因子的另一种表征。其证明利用了模运算符。

定理10.3 对于任意两个正整数 a 和 b , $\gcd(a, b)$ 是满足 $d = ia + jb$ 的最小正整数 d , 其中 i 和 j 是某些整数。换句话说, 如果 d 是 a 和 b 线性组合的最小正整数, 那么 $d = \gcd(a, b)$ 。

证明 假设 d 是满足 $d = ia + jb$ 的最小正整数, 其中 i 和 j 为整数。注意: 由 d 的定义直接可知 a 和 b 的任何公因子也是 d 的一个因子。因此, $d \geq \gcd(a, b)$ 。为了完成证明, 还要证明 $d \leq \gcd(a, b)$ 。

454

设 $h = \lfloor a/d \rfloor$, 即 h 是一个满足 $a \bmod d = a - hd$ 的整数。那么

$$\begin{aligned} a \bmod d &= a - hd \\ &= a - h(ia + jb) \\ &= (1 - hi)a + (-hj)b \end{aligned}$$

换句话说, $a \bmod d$ 也是 a 和 b 的一个整数线性组合。此外, 由模运算符的定义, $a \bmod d < d$ 。但 d 是 a 和 b 的最小正整数线性组合。因此, 必定可得 $a \bmod d = 0$, 这蕴涵着 $d|a$ 。此外, 通过类似的证明过程, 可得 $d|b$ 。因此, d 是 a 和 b 的一个因子, 这蕴涵着 $d \leq \gcd(a, b)$ 。■

如10.1.3节所示, 这个定理表明 \gcd 函数可用于计算模乘法逆元。在下一小节中, 将会表明如何快速计算 \gcd 函数。

10.1.2 欧几里得 GCD 算法

可以利用最古老的欧几里得算法, 计算两个数的最大公因子。这个算法基于 $\gcd(a, b)$ 的以下性质:

引理10.1 设 a 和 b 是两个正整数。对于任意整数 r , 有

$$\gcd(a, b) = \gcd(b, a - rb)$$

证明 设 $d = \gcd(a, b)$ 和 $c = \gcd(b, a - rb)$, 即 d 是满足 $d|a$ 和 $d|b$ 的最大整数, c 是满足 $c|b$ 和 $c|(a - rb)$ 的最大整数。我们要证明 $d = c$ 。由 d 的定义可知, 数

$$(a - rb)/d = a/d - r(b/d)$$

是一个整数。因此, d 整除 a 和 $(a - rb)$; 因此, $d \leq c$ 。

由 c 的定义可知, 由于 $c|b$, $k = b/c$ 必定是一个整数。此外, 数

$$(a - rb)/c = a/c - rk$$

也必定为整数, 这是因为 $c|(a - rb)$ 。因此, a/c 也必定是一个整数, 即 $c|a$ 。于是, c 整除 a 和 b ; 因此, $c \leq d$ 。综上可得 $d = c$ 。■

引理10.1容易导出一个称为欧几里得算法的古老算法, 可用于计算两个数的最大公因子 (GCD), 如算法10-1所示。

455

算法10-1 欧几里得GCD算法

```

算法 EuclidGCD( $a, b$ ):
    输入: 非负整数 $a$ 和 $b$ 
    输出: gcd( $a, b$ )
    if  $b = 0$  then
        return  $a$ 
    return EuclidGCD( $b, a \bmod b$ )

```

欧几里得算法的执行示例显示在表10-1中。

表10-1 欧几里得算法计算gcd(412, 260) = 4的执行过程示例。每次递归调用方法EuclidGCD(412, 260)的参数 a 和 b 从左到右显示, 其中列标题显示了EuclidGCD方法的递归层数

	1	2	3	4	5	6	7
a	412	260	152	108	44	20	4
b	260	152	108	44	20	4	0

1. 分析欧几里得算法

方法EuclidGCD(a, b)进行的算术操作数与递归调用次数成正比。为了限制欧几里得算法执行的算术操作数, 我们只需限制它的递归调用次数。首先, 观察到第一次调用之后, 第一个参数总是大于第二个参数。对于 $i > 0$, 设 a_i 是第 i 次递归调用方法EuclidGCD时的第一个参数。显然, 递归调用的第二个参数为 a_{i+1} , 即为下一次调用的第一个参数。同样, 可得

$$a_{i+2} = a_i \bmod a_{i+1}$$

这蕴涵着 a_i 的序列是严格递减的。我们将要证明序列递减得很快。特别声明

$$a_{i+2} < 1/2 a_i$$

分两种情况证明这个声明:

情况1: $a_{i+1} \leq 1/2 a_i$ 。因为 a_i 是严格递减的, 由此可得

$$a_{i+2} < a_{i+1} \leq 1/2 a_i$$

情况2: $a_{i+1} > 1/2 a_i$ 。在这种情况下, 由于 $a_{i+2} = a_i \bmod a_{i+1}$, 由此可得

$$a_{i+2} = a_i \bmod a_{i+1} = a_i - a_{i+1} < 1/2 a_i$$

456

因此, 随着每隔一次递归调用EuclidGCD方法, 它的第一个参数的大小都会减半。可以对上述分析概括如下:

定理10.4 设 a 和 b 是两个正整数。欧几里得算法计算gcd(a, b)需要执行 $O(\log \max(a, b))$ 次算术操作。

注意这里对算术操作数进行计数作为复杂度界限。事实上利用欧几里得算法可以在现代数字计算机上实现的事实, 可以改进上述界限中的常数因子。

2. 二进制欧几里得算法

欧几里得算法的一个变体, 称为二进制欧几里得算法 (binary Euclid's algorithm), 考虑如下事实, 在计算机上被2除的整数除法要比被一般整数除的除法执行得更快, 因为只要通过右移位 (right-shift) 这一本机处理器指令就可完成。二进制欧几里得算法如算法10-2所示。像原始欧几里得算法一样, 它用 $O(\log \max(a, b))$ 次算术操作计算两个整数 a 和 b 之间的最大公因子, 但其中的常数因子较小。该算法的正确性证明及其运行时间的渐近分析将在习题C-10.1中进行

探讨。

算法10-2 计算两个非负整数的最大公因子的二进制欧几里得算法

```

算法 EuclidBinaryGCD( $a, b$ ):
  输入: 非负整数 $a$ 和 $b$ 
  输出:  $\gcd(a, b)$ 
  if  $a = 0$  then
    return  $b$ 
  else if  $b = 0$  then
    return  $a$ 
  else if  $a$ 是偶数且 $b$ 是偶数 then
    return  $2 \cdot \text{EuclidBinaryGCD}(a/2, b/2)$ 
  else if  $a$ 是偶数且 $b$ 是奇数 then
    return  $\text{EuclidBinaryGCD}(a/2, b)$ 
  else if  $a$ 是奇数且 $b$ 是偶数 then
    return  $\text{EuclidBinaryGCD}(a, b/2)$ 
  else
    { $a$ 是奇数且 $b$ 是奇数}
    return  $\text{EuclidBinaryGCD}(|a - b|/2, \min(a, b))$ 

```

457

10.1.3 模运算

设 Z_n 表示小于 n 的非负整数集合:

$$Z_n = \{0, 1, \dots, (n-1)\}$$

集合 Z_n 也称为模 n 的剩余集合, 因为如果 $b = a \bmod n$, 那么有时称 b 为 $a \bmod n$ 的剩余。 Z_n 中的模运算是对于 Z_n 中的元素进行模 n 操作, 展示类似于那些传统运算的性质, 如结合律、交换律、加法和乘法的分配律以及加法和乘法分别存在恒等元0和1。此外, 在任何算术表达式中, 归约其每个子表达式模 n 的结果和计算整个表达式的结果相同。同样, 对于 Z_n 中的每个元素 x , 存在加法逆元 (additive inverse), 即对于每个 $x \in Z_n$, 存在 $y \in Z_n$, 满足 $x + y \bmod n = 0$ 。例如, $5 \bmod 11$ 的加法逆元是6。

但是, 当计算乘法逆元时, 就会出现重要差别。设 x 是 Z_n 中的元素。 x 的乘法逆元 (multiplicative inverse) 是元素 $x^{-1} \in Z_n$, 满足 $xx^{-1} \equiv 1 \bmod n$ 。例如, $5 \bmod 9$ 的乘法逆元是2, 即 $5^{-1} = 2$ (在 Z_9 中)。如在标准运算中那样, 0在 Z_n 中不存在乘法逆元。有趣的是, 某些非零元素在 Z_n 中也不存在乘法逆元。例如, 3在 Z_9 中不存在乘法逆元。然而, 如果 n 是素数, 那么 Z_n 中的每个 $x \neq 0$ 的元素在 Z_n 中都存在乘法逆元 (1是它自己的乘法逆元)。

定理10.5 当且仅当 $\gcd(x, n) = 1$ (即除1之外 x 和 n 没有公因子) 时, Z_n 中的元素 $x > 0$ 在 Z_n 中存在乘法逆元。

证明 假定 $\gcd(x, n) = 1$ 。由定理10.3可知, 存在整数 i 和 j , 满足 $ix + jn = 1$ 。这蕴涵着 $ix \bmod n = 1$, 即 $i \bmod n$ 是 x 在 Z_n 中的乘法逆元。这就证明了定理中的“当”部分。

为了证明“仅当”部分, 用反证法。假定 $x > 1$ 整除 n , 且存在满足 $xy \equiv 1 \bmod n$ 的元素 y 。对于某个整数 k , 有 $xy = kn + 1$ 。因此, 找到整数 $i = y$ 和 $j = -k$ 满足 $ix + jn = 1$ 。由定理10.3可知, 这蕴涵着 $\gcd(x, n) = 1$, 从而出现矛盾。 ■

如果 $\gcd(x, n) = 1$, 则称 x 和 n 是互素的 (relative prime) (1与所有其他整数都是互素的)。因

此, 定理10.5蕴涵着当且仅当 x 与 n 是互素的时, x 在 Z_n 中存在乘法逆元。此外, 定理10.5还蕴涵着序列 $0, x, 2x, 3x, \dots, (n-1)x$ 是 Z_n 中元素的一种简单重排序, 即它是 Z_n 中元素的一个排列, 如以下推论所示:

推论10.1 设 $x > 0$ 是 Z_n 中的元素, 满足 $\gcd(x, n) = 1$ 。那么

$$Z_n = \{ix: i = 0, 1, \dots, n-1\}$$

证明 见习题R-10.7。

458

在表10-2中, 显示了 Z_{11} 中元素的乘法逆元。当 x 在 Z_n 中的乘法逆元 x^{-1} 存在时, 表达式中的符号 y/x 取模 n 表示“ $yx^{-1} \bmod n$ ”。

表10-2 Z_{11} 中元素的乘法逆元

x	0	1	2	3	4	5	6	7	8	9	10
$x^{-1} \bmod 11$		1	6	4	3	9	2	8	7	5	10

1. 费马小定理

现在有证明第一个主要定理的足够工具, 称这个定理为费马小定理 (Fermat's Little Theorem)。

定理10.6 (费马小定理) 设 p 是一个素数, x 是一个满足 $x \bmod p \neq 0$ 的整数。那么

$$x^{p-1} \equiv 1 \pmod{p}$$

证明 对于 $0 < x < p$ 足以证明这个结果。因为

$$x^{p-1} \bmod p = (x \bmod p)^{p-1} \bmod p$$

因为可以化简“ $x^{p-1} \bmod p$ ”中的每个子表达式“ x ”。

由推论10.1可知, 对于 $0 < x < p$, 集合 $\{1, 2, \dots, p-1\}$ 和集合 $\{x \cdot 1, x \cdot 2, \dots, x \cdot (p-1)\}$ 恰好包含相同的元素。于是, 当把这些集合中的元素相乘在一起时, 得到同一个值, 即得到

$$1 \cdot 2 \cdots (p-1) = (p-1)!$$

换句话说,

$$(x \cdot 1) \cdot (x \cdot 2) \cdots (x \cdot (p-1)) \equiv (p-1)! \pmod{p}$$

如果把 x 项合并, 得

$$x^{p-1} (p-1)! \equiv (p-1)! \pmod{p}$$

因为 p 是素数, Z_p 中的每个非空元素都存在乘法逆元。因此, 从两边消去项 $(p-1)!$, 得到 $x^{p-1} \equiv 1 \bmod p$, 这正是想要的结果。

459

在表10-3中, 显示了 Z_{11} 中非空元素的幂。可观察到以下有趣的模式:

- 表的最后一列值为 $x^{10} \bmod 11$, 包含费马小定理中给出的所有值, 其中 $x = 1, \dots, 10$ 。
- 在第1行中, 一个元素(1)的子序列重复10次。
- 在第10行中, 以1结尾的两个元素的子序列重复5次, 因为 $10^2 \bmod 11 = 1$ 。
- 在第3、4、5和9行中, 以1结尾的5个元素的子序列重复2次。
- 在第2、6、7和8行中, 10个元素互不相同。
- 构成表中行的子序列的长度及其重复次数是10的约数, 即1、2、5和10。

表10-3 $Z_{11} \bmod 11$ 中元素的连续幂

x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^{10}
1	1	1	1	1	1	1	1	1	1
2	4	8	5	10	9	7	3	6	1
3	9	5	4	1	3	9	5	4	1
4	5	9	3	1	4	5	9	3	1
5	3	4	9	1	5	3	4	9	1
6	3	7	9	10	5	8	4	2	1
7	5	2	3	10	4	6	9	8	1
8	9	6	4	10	3	2	5	7	1
9	4	3	5	1	9	4	3	5	1
10	1	10	1	10	1	10	1	10	1

2. 欧拉定理

一个正整数 n 的欧拉 ϕ 函数 (totient function) 定义为小于或等于 n 且与 n 互素的正整数个数, 用 $\phi(n)$ 表示, 即 $\phi(n)$ 等于 Z_n 中存在乘法逆元的元素的个数。如果 p 是一个素数, 那么 $\phi(p) = p-1$ 。实际上, 因为 p 为素数, 那么数 $1, 2, \dots, p-1$ 与它互素, 且 $\phi(p) = p-1$ 。

如果 n 不是素数那又会怎样? 假定 $n = pq$, 其中 p 和 q 是素数。与 n 互素的数有多少个? 起初, 观察到 1 和 n 之间有 pq 个正整数。然而, 它们 (包含 n) 中有 q 个是 p 的倍数, 因而, 它们含有 p 与 n 的gcd。类似地, 有 p 个是 q 的倍数 (包含 n)。这些倍数不能计算在 $\phi(n)$ 内。因此, 有

$$\phi(n) = pq - q - (p-1) = (p-1)(q-1)$$

欧拉 ϕ 函数与 Z_n 的一个重要子集密切相关, 这个子集称为 Z_n 的乘法群 (multiplicative group), 用 Z_n^* 表示。集合 Z_n^* 定义为与 n 互素的 $1 \sim n$ 之间整数集合。如果 n 是素数, 则 Z_n^* 包含 Z_n 中的 $n-1$ 个非零元素, 即 $Z_n^* = \{1, 2, \dots, n-1\}$ 。一般而言, Z_n^* 包含 $\phi(n)$ 个元素。

集合 Z_n^* 具有几个有趣的性质。最重要的一个性质是在模 n 乘法之下, 这个集合是封闭的, 即对于 Z_n^* 中的任何元素 a 和 b , $c = ab \bmod n$ 也在 Z_n^* 中。实际上, 由定理10.5可知, Z_n 中存在 a 和 b 的乘法逆元。为了看到 Z_n^* 的这个封闭性质, 设 $d = a^{-1}b^{-1} \bmod n$ 。显然, $cd \bmod n = 1$, 它蕴涵着 d 是 c 在 Z_n 中的乘法逆元。因此, 再次应用定理10.5, 可得 c 与 n 互素, 即 $c \in Z_n^*$ 。利用代数术语, 称 Z_n^* 是一个群 (group), 这是说 Z_n^* 中的每个元素都存在乘法逆元, 并且 Z_n^* 中的乘法是可结合的, 存在恒等元且在 Z_n^* 中是封闭的一种简称。

Z_n^* 中有 $\phi(n)$ 个元素并且是一个乘法群的事实自然导致了对费马小定理的扩展。回忆在费马小定理中, 由于 p 是素数, 幂是 $p-1 = \phi(p)$ 。其结果是费马小定理的推广形式也为真。以下定理介绍了它的推广形式, 称为欧拉定理 (Euler's Theorem)。

定理10.7 (欧拉定理) 设 n 是正整数, x 是满足 $\gcd(x, n) = 1$ 的整数。那么

$$x^{\phi(n)} \equiv 1 \pmod{n}$$

证明 证明技术类似于费马小定理的证明技术。 $u_1, u_2, \dots, u_{\phi(n)}$ 表示 Z_n 的乘法群, 即集合 Z_n^* 中的元素。由 Z_n^* 的封闭性和推论10.1可知

$$Z_n^* = \{xu_i : i = 1, \dots, \phi(n)\}$$

即用 $x \bmod n$ 乘以 Z_n^* 中的元素只会改变序列 $u_1, u_2, \dots, u_{\phi(n)}$ 。因此, 把 Z_n^* 中的元素相乘在一起, 得

$$(xu_1) \cdot (xu_2) \cdots (xu_{\phi(n)}) \equiv u_1 u_2 \cdots u_{\phi(n)} \pmod{n}$$

再次把左边整理为一项 $x^{\phi(n)}$, 得到同余式

$$x(u_1 u_2 \cdots u_{\phi(n)}) \equiv u_1 u_2 \cdots u_{\phi(n)} \pmod{n}$$

两边同除以 u_i 的乘积, 得

$$x^{\phi(n)} \equiv 1 \pmod{n}$$

定理10.7给出了乘法逆元的封闭形式的表达式, 即如果 x 和 n 互素, 可写为

$$x^{-1} \equiv x^{\phi(n)-1} \pmod{n}$$

461

3. 生成元

给定素数 p 及 $1 \sim p-1$ 之间的整数 a , a 的阶为满足

$$a^e \equiv 1 \pmod{p}$$

的最小指数 $e > 1$ 。 Z_p 的一个生成元(generator)(也称为原根(primitive root))是 Z_p 中阶为 $p-1$ 的一个元素 g , 利用术语“生成元”表示这样一个元素 a , 是因为 a 的重复指数可以生成 Z_p^* 中的所有元素。例如, 如在表10-3中所示的那样, Z_{11} 的生成元是2、6、7和8。在许多计算中, 生成元起着非常重要的作用, 包括在10.4节讨论的快速傅里叶变换算法中。以下定理不加证明地建立了生成元的存在性。

定理10.8 如果 p 是一个素数, 那么集合 Z_p 中有 $\phi(p-1)$ 个生成元。

10.1.4 模指数运算

首先讨论模指数运算。在这种情况下, 主要问题是找到不同于显式蛮力方法的方法。但是, 在我们描述过一种有效算法之前, 先回顾一下这个朴素的算法, 因为这个算法中已经包含了一种重要的技术。

1. 蛮力指数运算

在任何指数算法中最重要的考虑之一是: 保持任何中间结果, 以避免变得过大。假定想要计算(比如) $30\,192^{43\,791} \bmod 65\,301$ 。使30 192自乘43 791次, 然后取模65 301的结果。在大多数程序设计语言中, 由于算术溢出, 这个计算将会产生难以预料的结果。因此, 应该在每次迭代中执行取模运算, 如算法10-3所示。

算法10-3 模指数运算的蛮力方法

算法 NaiveExponentiation(a, p, n):

输入: 整数 a 、 p 和 n

输出: $r = a^p \bmod n$

$r \leftarrow 1$

for $i \leftarrow 1$ **to** p **do**

$r \leftarrow (r \cdot a) \bmod n$

return r

这个“朴素”的指数算法是正确的, 但不太有效, 因为它需要 $\Theta(p)$ 次迭代, 以计算一个幂为 p 的数的模指数。对于较大的指数, 这个运行时间相当慢。幸运的是, 还有更好的方法。

2. 反复平方算法

改进的指数算法的一种简单而又重要的观察是对数 a^p 进行平方运算, 其结果与它的指数 p 乘以2等价。此外, 两个数 a^p 和 a^q 相乘等价于计算 $a^{(p+q)}$ 。于是, 把指数 p 写成二进制数 $p_{b-1} \cdots p_0$, 即

462

$$p = p_{b-1}2^{b-1} + \dots + p_02^0$$

当然, 每个 p_i 要么为0要么为1。利用上述观察, 通过计算多项式的Horner法则的一个变体, 可以计算 $a^p \bmod n$, 其中的多项式就是指数 p 的上述二进制表达式。确切地讲, 定义 q_i 为一个数, 该数的二进制表达式就是 p 的最左边的 i 位, 即 q_i 的二进制表示为 $p_{b-1}\dots p_{b-i}$ 。显然, 有 $p = q_b$ 。注意 $q_1 = p_{b-1}$, q_i 可以递归定义如下:

$$q_i = 2q_{i-1} + p_{b-i}, \text{ 其中 } 1 < i \leq b$$

因此, 用递归计算来求 $a^p \bmod n$ 的值, 称这种方法为反复平方方法 (repeated squaring), 如算法10-4所示。

算法 10-4 利用反复平方方法计算模指数的算法FastExponentiation。注意在方法FastExponentiation中, 由于在每次算术操作之后应用模运算符, 因此每个乘法操作和模操作中的操作数大小永远不会超过 $2\lceil \log_2 n \rceil$ 位

算法 FastExponentiation(a, p, n):

输入: 整数 a 、 p 和 n

输出: $r = a^p \bmod n$

if $p=0$ then

return 1

if p 是偶数 then

$t \leftarrow \text{FastExponentiation}(a, p/2, n)$ (p 是偶数, 于是 $t = a^{p/2} \bmod n$)

return $t^2 \bmod n$

$t \leftarrow \text{FastExponentiation}(a, (p-1)/2, n)$ (p 是奇数, 于是 $t = a^{(p-1)/2} \bmod n$)

return $a(t^2 \bmod n) \bmod n$

这个算法的主要思想是反复用2除 p , 直到 p 变为0, 在此过程中, 依次考虑指数 p 的每一位, 并对当前考虑的每个位的乘积 Q_i 取平方。此外, 如果当前位是1 (即 p 是奇数), 那么还在底数上乘以 a 。为了看出这个算法为什么起作用, 对于 $i = 1, \dots, b$, 定义

$$Q_i = a^{q_i} \bmod n$$

由 q_i 的递归定义, 导出以下 Q_i 的定义:

$$\begin{aligned} Q_i &= (Q_{i-1}^2 \bmod n) a^{p_{b-i}} \bmod n & \text{其中 } 1 < i \leq b \\ Q_1 &= a^{p_{b-1}} \bmod n \end{aligned} \quad (10.1)$$

463 容易验证 $Q_b = a^p \bmod n$ 。

表10-4中显示了模指数运算的反复平方算法的执行示例。

表10-4 模指数运算的反复平方算法的执行示例。对于每次递归调用FastExponentiation(2, 12, 13), 显示第二个参数 p , 以及输出值 $r = 2^p \bmod 13$

p	12	6	3	1	0
r	1	12	8	2	1

易于分析反复平方算法的运行时间。参照算法10-4, 不计递归调用的时间, 算法将执行算术操作的次数为常数。同时, 在每次递归调用中, 指数 p 减半。因此, 递归调用和算术操作的次数为 $O(\log p)$ 。概括如下:

定理10.9 设 a 、 p 和 n 是正整数, 且 $a < n$ 。反复平方算法计算 $a^p \bmod n$ 要利用 $O(\log p)$ 次算术操作。

10.1.5 模乘法逆元

现在转到计算 Z_n 中的乘法逆元的问题。首先,回忆定理10.5,它指出当且仅当 $\gcd(x, n) = 1$ 时, Z_n 中的非负元素 x 存在逆元,定理10.5的证明中实际上暗示一种计算 $x^{-1} \bmod n$ 的方法,即应该找到定理10.3中提到的整数 i 和 j ,满足

$$ix + jn = \gcd(x, n) = 1$$

如果能够找到这样的整数 i 和 j ,直接可得

$$i \equiv x^{-1} \bmod n$$

可用欧几里得算法的一个变体计算定理10.3中提到的整数 i 和 j ,这个算法称为扩展欧几里得算法(Extended Euclid's Algorithm)。

扩展欧几里得算法

设 a 和 b 是正整数,用 d 表示它们的最大公因子:

$$d = \gcd(a, b)$$

设 $q = a \bmod b$ 且 r 是一个整数,满足 $a = rb + q$,即

$$q = a - rb$$

欧几里得算法基于反复应用公式:

$$d = \gcd(a, b) = \gcd(b, q)$$

464

该公式直接由引理10.1可得。

假定算法递归调用时参数为 b 和 q ,还返回整数 k 和 l ,满足

$$d = kb + lq$$

回忆 r 的定义,有

$$d = kb + lq = kb + l(a - rb) = la + (k - lr)b$$

因此,有

$$d = ia + jb \quad \text{其中 } i = l \text{ 且 } j = k - lr$$

上一个方程暗示一种计算 i 和 j 的方法。这种方法称为扩展欧几里得算法,如算法10-5所示。表10-5中给出了该算法的一个执行示例。它的分析与欧几里得算法的分析类似。

算法10-5 扩展欧几里得算法

```

算法 ExtendedEuclidGCD( $a, b$ ):
  输入: 非负整数 $a$ 和 $b$ 
  输出: 整数三元组( $d, i, j$ ), 满足 $d = \gcd(a, b) = ia + jb$ 
  if  $b = 0$  then
    return ( $a, 1, 0$ )
   $q \leftarrow a \bmod b$ 
  设 $r$ 是一个整数, 满足 $a = rb + q$ 
  ( $d, k, l$ )  $\leftarrow$  ExtendedEuclidGCD( $b, q$ )
  return ( $d, l, k - lr$ )

```

定理10.10 设 a 和 b 是两个正整数。计算整数三元组(d, i, j)的扩展欧几里得算法满足

$$d = \gcd(a, b) = ia + jb$$

它执行 $O(\log \max(a, b))$ 次算术操作。

推论10.2 设 x 是 Z_n 中的一个元素, 满足 $\gcd(x, n) = 1$ 。计算 x 在 Z_n 中的乘法逆元需要 $O(\log n)$ 次算术操作。

表10-5 ExtendedEuclidGCD(a, b)的执行过程。 $a = 412$ 且 $b = 260$, 计算满足 $d = \gcd(a, b) = ia + jb$ 的 (d, i, j) 。对于每次递归调用, 显示参数 a 和 b 、变量 r 以及输出值 i 和 j 的值。输出值 d 总是 $\gcd(412, 260) = 4$

a	412	260	152	108	44	20	4
b	260	152	108	44	20	4	0
r	1	1	1	2	2	5	
i	12	-7	5	-2	1	0	1
j	-19	12	-7	5	-2	1	0

465

10.1.6 素性测试

素数在与数有关的计算包括密码计算中起着重要作用。但如何测试一个数 n 是否是素数, 尤其是在它很大时?

对于大数 n , 测试 n 的所有可能的因子在计算上是不可行的。而费马小定理(定理10.6)似乎提供了一种有效解法。也许可以以某种方式利用方程

$$a^{p-1} \equiv 1 \pmod{p}$$

构造 p 的一种测试方法, 即选择数 a 并把它提高到幂 $p-1$ 。如果结果不是(not) 1, 那么数 p 肯定不是素数。否则, 存在它是素数的机会。能够对于 a 的不同值重复这个测试, 以证明 p 是素数吗? 不幸的是, 答案是“不能”。存在称为卡迈克尔数(Carmichael number)的一类数, 它们具有性质 $a^{n-1} \equiv 1 \pmod{n}$, 其中 $1 \leq a \leq n-1$, 但 n 是合数。这些数的存在毁灭了上面提出的简单测试方法。卡迈克尔数的例子有561和1105。

1. 素性测试模板

当上述“随机”测试不起作用时, 通过更加复杂地利用费马小定理可得几种相关的测试。这些随机素性测试基于以下的一般方法。设 n 是我们想要测试其素性的奇整数, 并设 $\text{witness}(x, n)$ 是随机变量 x 和 n 的布尔函数, 它具有以下性质:

- (1) 如果 n 是素数, 那么 $\text{witness}(x, n)$ 总为假。因此, 如果 $\text{witness}(x, n)$ 为真, 那么 n 肯定为合数。
- (2) 如果 n 是合数, 那么 $\text{witness}(x, n)$ 为假的概率为 $q < 1$ 。

称函数 witness 为合性证据函数(compositeness witness function), 错误概率为 q 。因为 q 限制了 witness 不正确地把合数当成素数的概率。对于参数 x 的独立随机值, 通过反复计算 $\text{witness}(x, n)$, 可以任意小的错误概率确定 n 是否是素数。对于 k 个独立随机的 x 值, 当 n 是一个合数时, $\text{witness}(x, n)$ 不正确地返回“假”的概率为 q^k 。基于此观察的一般概率素性测试算法如算法10-6所示。这个算法利用称为模板方法模式的设计技术, 假设有一个满足上述两个条件的合性证据函数 witness 。为了把这个模板变成成熟的算法, 只需要指定如何选择随机数 x 和计算合性证据函数 $\text{witness}(x, n)$ 的细节。

466

算法10-6 基于合性证据函数 $\text{witness}(x, n)$ 的随机素性测试算法的模板。假设辅助方法 $\text{random}()$ 从随机变量 x 的域中随机选择一个值

算法 RandomizedPrimalityTesting(n, k):

输入: 奇整数 $n \geq 2$ 和信心参数 k

输出: 表明 n 是否是合数(总是正确的)或素数(不正确的错误概率为 2^{-k})的指示
{这个方法假设有一个错误概率为 $q < 1$ 的合性证据函数 $\text{witness}(x, n)$ }

```

 $t \leftarrow \lceil k/\log_2(1/q) \rceil$ 
for  $i \leftarrow 1$  to  $t$  do
     $x \leftarrow \text{random}()$ 
    if  $\text{witness}(x, n)$  then
        return “合数”
return “素数”

```

如果方法RandomizedPrimalityTesting($n, k, \text{witness}$)返回“合数”，可以肯定 n 是合数。然而，如果方法返回“素数”， n 实际上是合数的概率不超过 2^{-k} 。实际上，假定 n 是合数，但方法返回“素数”。可知对于 x 的 t 个随机值，证据函数 $\text{witness}(x, n)$ 计算为真。这个事件的概率为 q' 。由信心参数 k 、迭代次数 t 和方法中第一条声明建立的证据函数错误概率 q 之间的关系，可得 $q' \leq 2^{-k}$ 。模板方法RandomizedPrimalityTesting中的第二个参数 k 是一个信心参数（confidence parameter）。

2. Solovay-Strassen素性测试算法

Solovay-Strassen素性测试算法（Solovay-Strassen algorithm for primality testing）是模板方法RandomizedPrimalityTesting的特例。这个算法中使用的合性证据函数基于一些数论事实，回顾如下。

设 p 是一个奇素数。如果 Z_p 中的元素 $a > 0$ 是 Z_p 中的某个元素 x 的平方，即

$$a \equiv x^2 \pmod{p}$$

那么称 a 是 Z_p 中的一个二次剩余（quadratic residue）。

对于 $a \geq 0$ ，勒让德符号（Legendre symbol） $\left(\frac{a}{p}\right)$ 定义为：

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{如果 } a \bmod p \text{ 是一个二次剩余} \\ 0 & \text{如果 } a \bmod p = 0 \\ -1 & \text{其他情况} \end{cases}$$

467

不要把勒让德符号的表示法与除法操作相混淆。可以证明（见习题C-10.2）

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}$$

去掉 p 是素数的限制，可以对勒让德符号进行推广。设 n 是具有素因式分解的正奇整数：

$$n = p_1^{e_1} \cdots p_k^{e_k}$$

对于 $a \geq 0$ ，雅可比符号 $\left(\frac{a}{n}\right)$ 由以下方程定义：

$$\left(\frac{a}{n}\right) = \prod_{i=1}^k \left(\frac{a}{p_i}\right)^{e_i} = \left(\frac{a}{p_1}\right)^{e_1} \cdot \left(\frac{a}{p_1}\right)^{e_1} \cdots \left(\frac{a}{p_k}\right)^{e_k}$$

像勒让德符号一样，雅可比符号取值为0、1或-1。算法10-7显示了计算雅可比符号的递归方法。算法的正确性证明省略（见习题C-10.5）。

算法10-7 雅可比符号的递归计算

算法 Jacobi(a, b):
输入: 整数 a 和 b
输出: 雅可比符号 $\left(\frac{a}{b}\right)$ 的值

468

```

if a = 0 then
    return 0
else if a = 1 then
    return 1
else if a mod 2 = 0 then
    if (b2-1)/8 mod 2 = 0 then
        return Jacobi(a/2, b)
    else
        return -Jacobi(a/2, b)
else if (a-1)(b-1)/4 mod 2 = 0 then
    return Jacobi(b mod a, a)
else
    return -Jacobi(b mod a, a)

```

如果 n 是素数, 那么雅可比符号 $\left(\frac{a}{n}\right)$ 与勒让得符号相同。因此, 对于 Z_n 中的任何元素 a , 当 n 为素数时, 有

$$\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n} \quad (10.2)$$

如果 n 为合数, 可能存在一些 a 值, 使方程10.2仍然成立。因而, 如果满足方程10.2, 那么则称 n 是底数为 a 的欧拉伪素数 (Euler pseudo-prime)。以下引理不加证明地给出了产生合性证据函数欧拉伪素数的性质。

引理10.2 设 n 是一个合数。 Z_n 中至多存在 $(n-1)/2$ 个正 a 值, 满足 n 是底数为 a 的欧拉伪素数。

Solovay-Strassen素性测试算法利用以下合性证据函数:

$$\text{witness}(x, n) = \begin{cases} \text{false} & \text{如果 } n \text{ 是底数为 } x \text{ 的欧拉伪素数} \\ \text{true} & \text{其他情况} \end{cases}$$

其中 x 是一个随机整数, 且有 $1 < x \leq n-1$ 。由引理10.2可知, 这个函数的错误概率为 $q \leq 1/2$ 。Solovay-Strassen素性测试算法可以表示成模板方法RandomizedPrimalityTesting的一种特例 (算法10-6), 它在算法10-8中重新定义了辅助方法witness(x, n)和random()。

算法10-8 对算法RandomizedPrimalityTesting (算法10-6) 中的辅助方法进行特殊化, 得到Solovay-Strassen算法

```

算法 witness(x, n):
    return (Jacobi(x, n) mod n) ≠ FastExponentiation(x, (n-1)/2, n)

算法 random():
    return 1 ~ n-1 之间的一个随机整数

```

Solovay-Strassen算法的运行时间分析很简单。因为合性证据函数的错误概率不超过 $1/2$, 可以设 $q = 2$ 。这蕴涵着迭代次数等于信心参数 k 。在每次迭代中, 计算witness(x, n)需要 $O(\log n)$ 次算术操作 (见定理10.6和习题C-10.5)。概括如下:

定理10.11 给定一个正奇整数 n 和一个信心参数 $k > 0$, Solovay-Strassen算法进行 $O(k \log n)$ 次

算术操作, 以错误概率 2^{-k} 确定 n 是否是素数。

469

3. Rabin-Miller素性测试算法

现在描述Rabin-Miller素性测试算法 (Rabin-Miller algorithm for primality testing)。它基于费马小定理 (定理10.6) 及以下引理。

引理10.3 设 p 是一个大于2的素数。如果 x 是 Z_p 中的一个元素, 满足

$$x^2 \equiv 1 \pmod{p}$$

那么

$$x \equiv 1 \pmod{p}$$

或者

$$x \equiv -1 \pmod{p}$$

Z_n 中的一个非平凡单位平方根 (nontrivial square root of the unity) 被定义为一个整数 x , 其中 $1 < x < n-1$, 满足 $x^2 \equiv 1 \pmod{n}$ 。引理10.3指出上述如果 n 是素数, 那么 Z_n 中不存在非平凡单位平方根。

Rabin-Miller算法利用这个事实定义witness(x, n)函数如下:

算法 witness(x, n):

把 $n-1$ 写成 $2^k m$, m 是奇数

计算 $y \leftarrow x^m \bmod n$

if $y \equiv 1 \pmod{n}$ **then**

return false { n 可能为素数}

for $i \leftarrow 1$ **to** $k-1$ **do**

if $y \equiv -1 \pmod{n}$ **then**

return false { n 可能为素数}

$y \leftarrow y^2 \bmod n$

return true { n 肯定为合数}

正如在习题 (C-10.6) 中探讨的那样, 如果Rabin-Miller合性证据函数返回**true**, 那么 n 肯定为合数。Rabin-Miller合性证据算法返回**false**的错误概率由以下引理不加证明地给出。

引理10.4 设 n 是一个合数。 Z_n 中至多存在 $(n-1)/4$ 个正数 x , 满足Rabin-Miller合性证据函数witness(x, n)返回真。

总结如下。

定理10.12 给定一个正奇整数 n 和一个信心参数 $k > 0$, Rabin-Miller算法进行 $O(k \log n)$ 次算术操作, 以错误概率 2^{-k} 确定 n 是否是素数。

Rabin-Miller算法广泛用于实际的素数测试中。

470

4. 查找素数

素数测试算法可用于在给定范围内选择一个随机素数, 或者在预先指定的位数内选择素数。我们不加证明地探讨数论中的以下结果。

定理10.13 小于或等于 n 的素数个数 $\pi(n)$ 为 $\Theta(n/\ln n)$ 。事实上, 如果 $n \geq 17$, 那么 $n/\ln n < \pi(n) <$

$1.26n/\ln n$ 。

在上述定理中， $\ln n$ 是 n 的自然对数，即底数为 e 的 n 的对数，其中 e 是欧拉数，它是一个超越数，其前几个数字是2.71828182845904523536...

定理10.13的结论是：随机整数 n 是素数的概率为 $1/\ln n$ 。因此，为了找到具有给定位数 b 的一个素数，随机生成 b 位的奇数，并测试它们的素性，直至找到一个素数。

定理10.14 给定一个整数 n 和一个信心参数 k ，可以以错误概率 2^{-k} 随机选择一个 b 位的素数，这要进行 $O(kb)$ 次算术操作。

10.2 密码计算

在因特网上进行的大量活动是以电子方式进行的，如通信（电子邮件）、购物（Web商店）和金融交易（在线银行）这些活动。然而，因特网自身是一个不安全的传输网络：因特网上传的数据通过中间的几个称为路由器（router）的专用计算机，它们可以看到数据并可能修改它。

目前已经开发了各种密码技术支持在不安全的网络（如因特网）上进行通信。尤其是，密码研究已经开发了如下有用的密码计算：

- **加密/解密**：称要被传输的消息 M 为明文（plaintext），在通过网络发送之前，把它变换成不可识别的字符串 C ，称为密文（ciphertext）。称这种变换为加密（encryption）。密文 C 被收到之后，利用逆变换（依赖于其他秘密信息）把它转换回明文。称这个逆变换为解密（decryption）。加密过程的核心是对于想要把 C 变换回 M 的局外人，它在计算上应该是不可行的（由于不知道接收方具有的秘密信息）。
- **数字签名**：消息 M 的作者计算源于 M 的一条消息 S ，它是作者知道的秘密信息。如果另一方可以容易地验证只有 M 的作者能够在合理的时间内计算出 S ，则消息 S 是一个数字签名（digital signature）。

471

1. 将密码计算用于信息安全服务

有时将加密和数字签名的计算与本章将要讨论的其他密码计算结合起来。然而，上述两种技术已经足以支持引言中讨论的信息安全服务。

- **数据完整性**：计算消息 M 的一个数字签名 S ，不仅帮助我们确定 M 的作者，还验证 M 的完整性，因为对 M 的修改会产生不同的签名。于是，要进行数据完整性检查，可以进行一个验证测试，以证明 S 事实上是消息 M 的一个数字签名。
- **鉴别**：上述密码工具以两种可能方式用于鉴别中。在密码（password）鉴别模式中，用户在客户应用程序中键入一个用户-标识和密码，这个组合很快被加密，并发送给鉴别方。如果加密的用户-标识和密码组合与用户数据库中的信息匹配，那么这个个体就会被鉴别（并且数据库中从不以明文方式存储密码）。另一方面，鉴别方可以按照随机消息 M 的形式给用户发出一个挑战，用户必须很快地为鉴别进行数字签名。
- **授权**：给定一个鉴别模式，通过保持称为访问控制表（access control list）的表发布授权信息，该表中关联一些敏感数据或计算，它们应该只能被授权的个体访问。另一方面，对敏感数据或计算拥有授权的持有者可以对消息 C 进行数字签名，授权用户执行某些任务。例如，消息可能形如“I U.S. Corporation vice president give person x permission to access our fourth quarter earnings data.”。
- **私密性**：通过对敏感信息进行加密，使其对非授权代理保密。

- 不可抵赖性：如果使各方协商一个合约 M ，对消息进行数字签名，那么可有一种证明方式，表明各方已经看到并同意消息 M 中的内容。

本节简介了密码计算。常规的人名（如Alice、Bob和Eve）用于表示密码协议中涉及的各方。我们主要关注公钥密码学（public-key cryptography），它基于上一节中讨论的数论性质和算法。在介绍公钥密码学的概念之前，先简略讨论另一种加密方法。

472

10.2.1 对称加密模式

如上所述，密码学中的一个基本问题是私密性，即从Alice向Bob发送一条消息，并使第三方Eve不能截取消息的一个副本，来获得任何信息。此外，观察可知：通过加密模式（encryption scheme）或密码（cipher）可以达到私密性，被传输的消息 M 称为明文，消息在网络上被发送之前，被加密成一种不可识别的字符串 C ，称为密文。在密文 C 被接收到之后，利用称为解密的逆变换把密文变回明文 M 。

1. 密钥

在描述加密模式的细节时，必须解释需要的所有步骤，以便将明文 M 加密成密文 C ，然后把密文解密回明文 M 。此外，为了使得Eve不能从 C 中提取 M ，必定存在某些秘密信息对她保密。

在传统密码学中，Alice和Bob共享公共密钥（secret key） k 。并且用它对消息加密和解密。这样的模式也称为对称密钥（symmetric encryption）模式，因为 k 既用于加密也用于解密，且Alice和Bob共享同一秘密。

2. 代换密码

对称密钥的一个经典例子是代换密码（substitution cipher），密钥是字母表上字符组成的一个置换 π 。把明文 M 加密变成密文 C ，就是用字符 $y = \pi(x)$ 代替 M 中的每个字符 x 。知道置换函数 π ，就很容易解密。可用字符 $x = \pi^{-1}(y)$ 代替 C 中的字符 y 导出 M 。凯撒密码（Caesar cipher）是代换密码的一个早期例子，其中用字符

$$y = x + k \bmod n$$

代换每个字符 x ，其中 n 是字母表大小， k 是密钥，并且 $1 < k < n$ 。称这种代替模式为“凯撒密码”，据说Julius Caesar用 $k = 3$ 作为密钥。

代换密码非常易于使用，但它们是不安全的。实际上，基于文本语言中各种字符出现的频率或者连续字符分组的知识，利用频率分析（frequency analysis）很快可以推导出密钥。

473

3. 一次一密乱码本

存在安全对称密码。实际上，对称密码是已知最安全的密码。它是一种一次一密乱码本。在这个密码系统中，Alice和Bob都共享一个与希望通信的消息一样大的随机位串 K 。串 K 是对称密钥，因为要从消息 M 计算出密文 C ，Alice计算

$$C = M \oplus K$$

其中 \oplus 表示按位异或运算符。即使Eve正在窃听，Alice也可以利用可靠的通信信道向Bob发送 C 。因为从随机串计算出密文 C 在计算上有难度。然而，Bob可以通过计算 $C \oplus K$ 容易地解密密文消息 C ，因为

$$\begin{aligned} C \oplus K &= (M \oplus K) \oplus K \\ &= M \oplus (K \oplus K) \\ &= M \oplus 0 \\ &= M \end{aligned}$$

其中 0 表示与 M 同长度的全 0 位串。这种模式显然是一个对称密码系统，因为密钥 K 用于加密和解密。

一次一密乱码本在计算上是有效的，因为按位异或运算是计算机执行得最快的操作之一。同样，正如已经提到的那样，一次一密乱码本非常安全。然而，一次一密乱码本密码系统应用不很广泛。这个系统的主要问题是Alice和Bob必须共享一个非常大的密钥。一次一密乱码本的安全主要取决于一个事实，密钥 K 只能用一次。如果 K 被重用，则存在能够破解这个系统的几种密码分析方法。对于实际的密码系统，我们希望密钥可以重用，并且要比加密和解密的消息小。

4. 其他对称密码方法

安全且有效的对称密码方法确实存在。这些对称密码方法的名称是只取首字母的缩写词或者有趣的名字，如“3DES”、“IDEA”、“Blowfish”和“Rijndael”（读作“Rhine-doll”）。它们对明文中的位进行一系列复杂代替和置换变换。虽然在许多应用中这些系统很重要，但从算法的角度讲，它们只能引起一般的兴趣；因此，介绍它们超出了本书的范围。它们的运行时间与加密和解密的消息长度成正比。因此，我们提及这些算法存在且快速。但是，在本书中，我们并不详细讨论这些有效的对称密码方法。

474

10.2.2 公钥密码系统

对称密码的主要问题是密钥传输（key transfer）问题，即如何分发加密和解密的密钥。1976年，Diffie和Hellman描述了一种抽象系统——公钥加密系统（public-key cryptosystem），可以避免这些问题。他们实际上并没有公布一个特殊的公钥系统，而是讨论了这种系统的特点。确切地讲，给定消息 M 、加密函数 E 、解密函数 D ，以下四个性质必定成立：

- (1) $D(E(M)) = M$ 。
- (2) E 和 D 都易于计算。
- (3) 从 E 导出 D 在计算上是不可行的¹。
- (4) $E(D(M)) = M$ 。

回顾一下，这些性质似乎具有相当普遍的意义。性质(1)只指出，一旦加密一条消息，应用解密过程恢复它。也许性质(2)更显而易见。要是密码系统切实可行，加密和解密在计算上必定快速。

性质(3)是创新的开始。它意味着 E 只有一种方法；逆转 E 在计算上是不可行的，除非你已经知道 D 。因此，加密过程 E 可以公开。任何一方都可以发送消息，但只有一方知道如何解密。

如果性质(4)成立，那么映射是一一对应关系。因此，密码系统是数字签名问题的一种解决方案。给定一则从Bob到Alice的电子消息，如何证明实际上是Bob发送了它？Bob可以把他的解密过程应用到某些签名消息 M 。然后任何其他一方可以应用公开加密过程 E ，验证Bob实际上发送了那条消息。因为只有Bob知道解密函数，只有Bob能够产生可由函数 E 正确解密的一个数字签名。

公钥密码学是现代密码学的基础。它在经济上的重要性正在快速增长，因为它提供了因特网上进行所有电子交易的安全基础设施。

公钥密码系统的设计可用一般意义的术语进行描述。其思想是在计算机科学中找到一个非常难的问题，然后把它与密码系统结合在一起。理想情况下，经过证明破解密密码系统在计算上等价于求解这个非常难的问题。有一大类这样的问题，称它们为NP完全问题，这些问题没有已知的多项式时间的解（见第13章）。事实上，普遍认为这些问题不存在多项式时间的解。为了产生特

1. 第13章正式介绍了计算难度的概念。

殊的加密密钥和解密密钥, 建立一个针对这个问题的参数的特殊集合。然后, 加密意味着把消息变成问题的一个实例。接收方可以毫不费力地利用秘密信息(解密密钥)解这个难题。

475

10.2.3 RSA 密码系统

必须关注的问题是如何把一个计算上的难题结合进密码系统中。早期的公钥密码系统之一(Merkle-Hellman系统)把加密与背包问题(knapsack problem)联系起来, 它是一个NP完全问题。不幸的是, 系统产生的问题是背包问题的一个特殊子类, 可容易地求解。因此, 设计公钥密码系统具有微妙之处。

可能最知名的公钥密码系统也是最古老的密码系统之一, 它把大数因式分解的难度结合进来。以它的发明者Rivest、Shamir和Adleman将其命名为RSA。

在这个密码系统中, 首先选择两个大素数 p 和 q , 设 $n=pq$ 是其乘积, 回忆 $\phi(n)=(p-1)(q-1)$ 。选择加密密钥和解密密钥 e 和 d 满足

- e 和 $\phi(n)$ 是互素的。
- $ed \equiv 1 \pmod{\phi(n)}$ 。

第二个条件意味着 d 是 $e \bmod \phi(n)$ 的乘法逆元。 n 和 e 的值对构成公钥, 其中 d 是私钥。在实际中, 选择 e 为随机数, 或为以下数之一: 3、17或65 537。

用RSA进行加密和解密的规则是简单的。为简单起见, 假设明文是一个整数 M , 且 $0 < M < n$ 。如果 M 是一个串, 可把它看作连接它的字符各位的一个整数。利用加密密钥 e 作为指数:

$$C \leftarrow M^e \bmod n \quad (\text{RSA加密})$$

应用模指数运算将明文 M 加密成密文 C 。

现在利用解密密钥 d 作为指数:

$$M \leftarrow C^d \bmod n \quad (\text{RSA解密})$$

应用模指数运算将密文 C 解密成明文 M 。

上述加密和解密规则的正确性由以下定理证实。

定理10.15 设 p 和 q 是两个奇整数, 定义 $n=pq$ 。设 e 与 $\phi(n)$ 互素, d 是 $e \bmod \phi(n)$ 的乘法逆元。对于满足 $0 < x < n$ 的每个整数 x ,

$$x^{ed} \equiv x \pmod{n}$$

证明 设 $y = x^{ed} \bmod n$ 。想要证明 $y = x$ 。根据选择 e 和 d 的方式, 对于某个整数 k , 可以有 $ed = k\phi(n) + 1$ 。因此, 有

$$y = x^{k\phi(n)+1} \bmod n$$

分两种情况。

情况1: x 并不整除 n 。重写 y 如下:

$$\begin{aligned} y &= x^{k\phi(n)+1} \bmod n \\ &= x x^{k\phi(n)} \bmod n \\ &= x(x^{\phi(n)} \bmod n)^k \bmod n \end{aligned}$$

由定理10.7(欧拉定理)可知, $x^{\phi(n)} \bmod n = 1$, 这蕴涵着 $y = x \cdot 1^k \bmod n = x$

情况2: x 整除 n 。因为 $n = pq$ 且 p 和 q 互素, x 是 p 或 q 的倍数。假定 x 是 p 的倍数, 即对于某个正

476

整数 h , $x = hp$ 。显然, x 不能同时是 q 的倍数。否则 x 就会大于 $n = pq$, 这会出现矛盾。因此, $\gcd(x, q) = 1$, 且由定理10.7 (欧拉定理) 可知,

$$x^{\phi(q)} \equiv 1 \pmod{q}$$

因为 $\phi(n) = \phi(p)\phi(q)$, 把上面的同余式两边的指数同时变为 $k\phi(q)$, 可得

$$x^{k\phi(n)} \equiv 1 \pmod{q}$$

将其重写为

$$x^{k\phi(n)} = 1 + iq$$

其中 i 为某个整数。将上面方程的两边同乘以 x , 并回忆 $x = hp$ 和 $n = pq$, 可得

$$\begin{aligned} x^{k\phi(n)+1} &= x + xiq \\ &= x + hpiq \\ &= x + (hi)n \end{aligned}$$

因此, 有

$$y = x^{k\phi(n)+1} \bmod n = x$$

不论哪一种情况, 都有 $y = x$, 这就证明了定理。 ■

1. 利用RSA进行数字签名

加密函数和解密函数的对称性意味着RSA密码系统直接支持数字签名。实际上, 把解密函数应用到消息 M 上, 得到消息 M 的一个数字签名 S , 即

$$S \leftarrow M^d \bmod n \quad (\text{RSA签名})$$

现在用加密函数验证数字签名 S , 即检查

$$M \equiv S^e \pmod{n} \quad (\text{RSA验证})$$

477

2. 破解RSA的难度

注意, 即使知道 e 的值, 也不能计算出 d , 除非知道 $\phi(n)$ 。大多数密码研究人员普遍认为破解RSA需要计算出 $\phi(n)$, 这就需要对 n 进行因式分解。尚未证明 (proof) 因式分解在计算上有难度, 在过去的几百年中, 许许多多的著名科学家从事这个问题的研究。尤其是当 n 很大时 (约200个数字), 因式分解要花很长的时间。为给你提供最新的认识, 当全国范围的计算机网络完成第九个费马数 $2^{512}-1$ 的因式分解时, 数学家们十分激动。这个数“只有”155位十进制数。除非有突破性的进展, RSA系统仍然安全。如果由于技术进步, 使得能够对200位的数进行因式分解, 就只需要选择300或400位的 n 。

3. 分析和创建RSA加密

RSA加密、解密、签名和验证的运行时间易于分析。实际上, 每个这样的操作都需要常数个模指数运算, 可用方法FastExponentiation (算法10-4) 完成。

定理10.16 设 n 是RSA密码系统中使用的模数。RSA加密、解密、签名和验证中的每个操作都需要 $O(\log n)$ 次算术操作。

为了创建RSA密码系统, 需要产生公钥和私钥对, 即需要计算伴随它的私钥 (d, p, q) 和公钥 (e, n) 。这涉及以下计算:

- 随机选择具有给定位数的两个素数 p 和 q 。通过测试随机整数的素性就可做到, 如在10.1.6节末尾讨论的那样。

- 选择与 $\phi(n)$ 互素的整数 e 。通过选择小于 $\phi(n)$ 的随机素数，直至找到一个不能整除 $\phi(n)$ 的数即可。在实际中，在一个给定的素数表中，检查小素数即可（通常 $e=3$ 或 $e=17$ 即可）。
- 计算 e 在 $Z_{\phi(n)}$ 中的乘法逆元 d ，利用扩展欧几里得算法可以做到（推论10.2）。

在这一章里，我们已经解释了这些数论问题中的算法。

478

10.2.4 El Gamal 密码系统

已经看到RSA密码系统的安全与大数因式分解的难度有关。可能构造一个基于其他数论难解问题的密码系统。现在考虑El Gamal密码系统，它以其发明者Taher El Gamal的名字命名，这个密码系统基于称为“离散对数”问题的难度。

1. 离散对数

当处理实数时， $\log_b y$ 为满足 $b^x = y$ 的 x 值。可以定义一个类似的离散对数。给定两个整数 b 和 n ，且 $b < n$ ，以 b 为底数的整数 y 的离散对数（discrete logarithm）是一个整数 x ，满足

$$b^x \equiv y \pmod{n}$$

离散对数也称为下标（index），记作

$$x = \text{ind}_{b,n} y$$

虽然把数提高到 $\text{mod } p$ 的大幂（回忆反复平方算法，算法10-4）非常有效，但是计算离散对数的逆元更困难。El Gamal系统依靠这个计算的难度。

2. El Gamal加密

设 p 是一个素数， g 是一个 Z_p 的生成元。私钥 x 是一个介于 $1 \sim p-2$ 之间的整数。设 $y = g^x \pmod{p}$ 。El Gamal加密公钥是一个三元组 (p, g, y) ，如果取如同公认那样难度的离散对数，则 $y = g^x \pmod{p}$ 并不能解出 x 。

为了对明文 M 加密，选择与 $p-1$ 互素的随机整数 k ，并计算以下数值对：

$$\begin{aligned} a &\leftarrow g^k \pmod{p} \\ b &\leftarrow My^k \pmod{p} \end{aligned} \quad (\text{El Gamal加密})$$

密文 C 由上述计算的数对 (a, b) 组成。

3. El Gamal解密

为了检索明文 M ，在El Gamal模式中对密文 $C = (a, b)$ 进行的解密很简单：

$$M \leftarrow b/a^x \pmod{p} \quad (\text{El Gamal解密})$$

479

在上述表达式中，用 a^x “除”可以解释为模运算中的内容，即用 a^x 在 Z_p 中的逆元乘以 M 。El Gamal加密模式的正确性容易验证。确实有

$$\begin{aligned} b/a^x \pmod{p} &= My^k(a^x)^{-1} \pmod{p} \\ &= Mg^{xk}(g^{kx})^{-1} \pmod{p} \\ &= M \end{aligned}$$

4. 利用El Gamal进行数字签名

上述模式的一种变体提供了一种数字签名方法，即选择与 $p-1$ （当然等于 $\phi(p)$ ）互素的一个随机整数 k ，并计算

$$\begin{aligned} a &\leftarrow g^k \bmod p \\ b &\leftarrow k^{-1}(M - xa) \bmod (p-1) \end{aligned} \quad (\text{El Gamal 签名})$$

得到消息 M 的一个签名, 它是一个数对 $S = (a, b)$,

为了验证数字签名 $S = (a, b)$, 检查

$$y^a a^b \equiv g^M \pmod{p} \quad (\text{El Gamal 验证})$$

El Gamal数字签名模式的正确性如下可见:

$$\begin{aligned} y^a a^b \bmod p &= ((g^x \bmod p)^a \bmod p) ((g^k \bmod p)^{k^{-1}(M-xa) \bmod (p-1)} \bmod p) \\ &= g^{xa} g^{kk^{-1}(M-xa) \bmod (p-1)} \bmod p \\ &= g^{xa+M-xa} \bmod p \\ &= g^M \bmod p \end{aligned}$$

5. El Gamal加密分析

El Gamal密码系统的性能分析类似于RSA密码系统的性能分析, 即有如下结论。

定理10.17 设 n 是El Gamal密码系统中使用的模数。El Gamal加密、解密、签名和验证中的每个操作都需要 $O(\log n)$ 次算术操作。

10.3 信息安全算法和协议

一旦有一些工具, 如涉及数的一些基本算法和公钥加密方法, 就可以开始把它们与其他算法结合起来, 提供所需的信息安全服务。本节讨论几个这样的协议, 有些协议除了利用上述讨论的算法, 还利用了以下讨论的主题。

10.3.1 单向散列函数

公钥密码系统常与单向散列函数 (one-way hash function) 联合使用。单向散列函数也称为消息摘要 (message digest) 或者指纹 (fingerprint)。下面给出这种函数的一种非形式描述。其形式化讨论超出了本书的范围。

单向散列函数 H 把一个任意长的串 (消息) M 映射为一个具有固定位数的整数 $d = H(M)$ 。这个整数称为 M 的摘要 (digest), 满足以下性质:

- (1) 给定一个串 M , 可以很快计算出 M 的一个摘要。
- (2) 给定 M 的摘要 d , 但不给出 M , 通过它找到 M 在计算上是不可行的。

给定一个串 M , 如果用同一摘要找到另一个串 M' 在计算上是不可行的, 则称这个单向散列函数是抗冲突的 (collision-resistant)。如果用同一摘要找到两个串 M_1 和 M_2 在计算上是不可行的, 则称这个单向散列函数是强抗冲突的 (strongly collision-resistant)。

有几个函数被认为是强抗冲突的, 单向散列函数已设计出。实际中应用最多的是MD5和SHA-1, MD5可产生128位的摘要, SHA-1可产生160位的摘要。现在考察单向散列函数的一些应用。

单向函数的一个应用是加速数字签名的构造过程。如果有一个抗冲突的单向散列函数, 则可以对消息摘要进行签名, 而不用对消息自身进行签名, 即签名 S 为

$$S = D(H(M))$$

除了小消息之外,实际中散列消息并对摘要进行签名要比直接对消息签名更快。同样,这个过程克服了RSA和El Gamal签名模式的一个重大局限性,即消息 M 必须小于模数 n 。例如,利用MD5时,只需利用一个大于 2^{128} 的固定模数 n ,就可以对任意长度的消息签名。

481

10.3.2 时间戳和认证字典

下一个应用是时间戳(timestamping)。Alice有一个文档 M ,她想得到文档 M 在当前时间 t 存在的证书。

在时间戳问题的一种方法中,Alice利用可信第三方Trevor的服务,来提供时间戳。Alice可以向Trevor发送消息 M ,让他签署一个由 M 和 t 的连接组成的新文档 M' 。当这种方法有效时,它还有一个缺点,就是Trevor可以看见 M 。一个抗冲突的单向散列函数 H 可以消除这个问题。Alice利用 H 计算 M 的摘要 d ,要求Trevor签名一个新消息 M'' ,它由 d 和 t 的连接组成。

1. 认证字典

事实上,可以定义时间戳问题的另一种方法。这种方法不需要完全信任Trevor。这种替代方法基于认证字典(authenticated dictionary)的概念。

在一个认证字典中,第三方Trevor收集数据项的字典数据库。在时间戳应用中,数据项是文档的摘要,需要在它们存在的某个日期加上时间戳。在这种情况下,我们并不信任Trevor关于Alice的文档 M 的摘要 d 在日期 t 存在的签名声明。代之以Trevor计算整个字典的摘要 D ,并把 D 公布在时间戳不会受到怀疑的地方(如著名报纸的分类专栏)。此外,Trevor用部分摘要 D' 响应Alice, D' 汇总了字典中所有数据项的摘要,除了Alice文档的摘要 d 之外。

要使这个模式高效工作,应该存在一个函数 f ,满足 $D = f(D', d)$,对于Alice而言, f 易于计算。但这个函数在某种意义上应该是单向的,即对于Trevor而言,给定任意 y ,如果要计算出满足 $D = f(x, y)$ 中的 x ,应该在计算上有难度。给定这样一个函数,可以依靠Trevor计算出他所接收的所有文档的摘要,并把摘要发布在公开位置。由于在计算上不可行,Trevor不可能伪造一个响应,表明值 d 已在字典中,而实际上并非如此。因此,这个协议的关键部分是存在单向函数 f 。在本节其余部分中,将探索一种构造适合认证字典的函数 f 的可能方法。

2. 散列树

一种称为散列树(hash tree)模式的有趣的数据结构方法可用于实现认证字典。这种结构支持字典数据库的初始构造,后接对每一数据项的查询操作或成员响应。

482

集合 S 的一棵散列树 T 把 S 中的每个元素存储在完全二叉树 T 的外部结点上,并在每个结点 v 处存储一个散列值 $h(v)$,它利用一个著名的单向函数把其子结点的散列值组合起来。在时间戳应用中,存储在 T 的外部结点上的数据项是它们自己文档的摘要,这些摘要在存在的特定日期被加上时间戳。 S 的认证字典由散列树 T 加上存储在 T 的根 r 中的值 $h(r)$ 的发布组成。在 T 中从存储 x 的结点到根的路径上,通过报告出这条路径的结点中存储的值以及在这条路径上具有兄弟结点的所有结点的值,证明元素 x 属于 S 。

给定这样一条路径 p ,Alice可以重新计算根的散列函数值 $h(r)$ 。然而,由于 T 是一棵完全二叉树,她计算这个值只需要调用 $O(\log n)$ 次散列函数 h ,其中 n 是 S 中的元素个数。

10.3.3 硬币抛掷和比特承诺

现在介绍一种协议,它允许Alice和Bob在网络上通过交换邮件消息或者其他通信,随机抛掷硬币。设 H 是一个强抗冲突的单向散列函数。Alice和Bob之间的交互作用由以下步骤组成:

- (1) Alice挑选一个数 x ，并计算摘要 $d = H(x)$ ，把 d 发给Bob。
- (2) Bob接到 d 之后，向Alice发送 x 是奇数还是偶数的猜测。
- (3) Alice宣布硬币抛掷的结果：如果Bob猜测正确，结果是正面；否则，是反面。她也给Bob发送 x 作为结果的证明。
- (4) Bob验证Alice没有欺骗他，即 $d = H(x)$ 。

强抗冲突要求是必要的，因为Alice可能给出两个数，一个是奇数，一个是偶数，具有相同的数字签名 d 。这样能够控制硬币抛掷的结果。

与硬币抛掷相关的是比特承诺 (bit commitment)。在这种情况下，Alice想要托付一个值 n (可能是一个比特，也可能是任意串)，同时该值对Bob保密。一旦Alice公开 n ，Bob希望验证她没有欺骗。例如，Alice想要向Bob证明，她可以预测某只股票明天会上升 ($n = 1$)、下降 ($n = -1$)，或者不变 ($n = 0$)。利用强抗冲突的单向散列函数 H ，协议进行如下：

(1) 她向Bob发送 x ，还有 x 、 y 和 n 连接的摘要。与经典密码文献中的表示一致，本章中“ $a||b$ ”表示串 a 和串 b 的连接。因此，利用这种表示方法，Alice向Bob发送 $d = H(x||y||n)$ 。注意，Bob不能由 x 和 d 计算出 n 。

(2) 在第二天交易结束之后， n 为公众所知，Alice向Bob发送 y 进行验证。

483

(3) Bob验证Alice没有欺骗，即 $d = H(x||y||n)$ 。

10.3.4 安全电子传输 (SET) 协议

最后一个应用要复杂得多，它涉及加密、数字签名和单向散列函数的组合应用。实际上它是因特网上安全信用卡支付的一个简化SET (secure electronic transaction) 版本。

Alice想要利用Lisa (一家银行) 颁发的信用卡从Barney (一个因特网书店) 购买一本书。Alice关注私有性：一方面，她不希望Barney看到她的信用卡号；另一方面，她不希望Lisa知道她从Barney买的是哪本书。然而，她希望Barney把书籍发给她，并且Lisa把付款发送给Barney。最后，Alice还想保证即使某个人在因特网上窃听，Barney、Lisa和她自己之间的通信总是私密的。算法10-9描述了利用一个强抗冲突的单向散列函数 H 的协议。

算法10-9 SET协议的简化版本

- (1) Alice准备两份文档，一份是订购单 O ，表明她想要从Barney订购的书；一份是支付表 P ，为Lisa提供交易中使用的卡号以及支付的金额。Alice计算摘要

$$o = H(O)$$

$$p = H(P)$$

并产生 o 和 p 连接的摘要的一个数字签名 S ，即

$$S = D_A(H(o||p)) = D(H(H(O)||H(P)))$$

其中 D_A 是Alice基于她的私钥签名使用的函数。Alice用Lisa的公钥对 o 、 P 和 S 的连接进行加密，产生密文

$$C_L = E_L(o||P||S)$$

她还用Barney的公钥对 O 、 p 和 S 的连接进行加密，产生密文

$$C_B = E_B(O||p||S)$$

她把 C_L 和 C_B 发给Barney。

- (2) Barney用他的私钥解密 C_B ，重新得到 O 、 p 和 S 。他用Alice的公钥，通过检查

$$E_A(S) = H(H(O)||p)$$

验证订购单 O 的真实性, 并把 C_L 转发给Lisa。

(3) Lisa用她的私钥解密 C_L , 重新得到 o 、 P 和 S 。她用Alice的公钥, 通过检查

$$E_A(S) = H(o||H(P))$$

验证支付表 P 的真实性, 且验证 P 指定给Barney付款。她建立了由交易号、Alice的名字和同意支付的金额组成的授权消息 M 。Lisa计算 M 的签名 T , 并把用Barney的公钥加密的数对 (M, T) 发送给Barney, 即 $C_M = E_B(M||T)$ 。

(4) Barney用Lisa的公钥解密 C_M , 重新得到 M 和 T , 并检查 $E_L(T) = M$ 来验证授权消息 M 的真实性。他验证 M 中的名字是Alice, 金额就是书的价格。他把书发给Alice, 并用Lisa的公钥加密向她发送的交易数, 要求Lisa支付, 完成订购过程。

(5) Lisa利用Alice的信用卡账户向Barney支付。

SET协议的性质

以下观察表明该协议满足私密性、完整性、不可抵赖性和可鉴别性。

- Barney不能看见Alice的信用卡号, 它存储在支付表 P 中。Barney有 P 的摘要 p 。然而, 他不能从 p 计算出 P , 因为 H 是单向的。Barney还有包含 P 的消息的密文 C_L 。然而, 由于Barney没有Lisa的私钥, 他不能解密 C_L 。
- Lisa不能看见Alice订购的书, 它存储在订购单 O 中。Lisa有 O 的一个摘要 o 。然而, 她不能从 o 计算出 O , 因为 H 是单向的。
- Alice提供的数字签名 S 有两个作用。它允许Barney验证Alice的订购单 O 的真实性, 允许Alice验证Alice的支付表 P 的真实性。
- Alice不能否认她订购了 O 中指定的那本书, 并且已用信用卡支付了 P 中指定的给定金额。的确, 因为 H 是抗冲突的, 她不可能伪造一份散列到同一摘要 o 和 p 的不同订购单和/或支付表。
- 各方之间的所有通信利用公钥加密, 即使出现窃听者, 也可保证私密性。

因此, 尽管SET协议有点复杂, 但它说明了如何组合密码计算进行一个有价值的电子商务操作。

484
485

10.3.5 密钥分发和交换

公钥密码系统假设任何一方都知道公钥。例如, 如果Alice想要向Bob发送一条私密消息, 她需要知道Bob的公钥。类似地, 如果Alice想要验证Bob的数字签名, 她也需要Bob的公钥。Alice如何得到Bob的公钥呢? Bob只能把它发送给Alice, 但如果Eve可以拦截Bob和Alice之间的通信, 就可以用她自己的密钥代替Bob的公钥, 因此, 欺骗Alice显示她的消息将发送给Bob, 或者相信Bob已经签名消息, 而实际上消息是由Eve签名的。

1. 数字证书

这种问题的一种解决方案是需要引入第三方Charlie, 他受到协议中所有参与者的信任。进一步假设每个参与者都有Charlie的公钥。Charlie向每个参与者颁发一个证书 (certificate), 它是Charlie以数字方式签署的一个声明, 包含参与者的名字及其公钥。Bob现在向Alice发送Charlie颁发给他的证书, Alice从证书中提取Bob的公钥, 并用Charlie的公钥 (回忆可知假设每个参与者都有Charlie的公钥) 验证它的真实性。数字证书广泛应用于实际的公钥应用中。在X.509 ITU (International Telecommunication Union) 标准中描述了它们的格式。除了证书及其公钥方面的主题之外, 证书还包括唯一的序列号和终止期。称证书的发布者证书颁发机构 (certificate

authority, CA)。

在现实中,通过引入分发公钥的证书,对前一节中描述的协议进行修改。同时,应该利用CA的公钥验证证书的有效性。

2. 证书撤销

私钥有时会丢失、失窃或者遭受其他危害。当发生这种情况时,CA应该撤销那个密钥的证书。例如,Bob可能在他的笔记本计算机的文件中保存他的私钥。如果这台计算机失窃,Bob应该请求CA立即撤销他的证书,因为窃贼可能模仿Bob。CA周期性地公布证书撤销表(certificate revocation list, CRL),它由所有未到期证书的序列号的一个签名表组成,这些证书将会与时间戳一起被撤销。

在验证证书的有效性时,参与者也应该从CA得到最新的CRL,并验证证书还未被撤销。CRL的时间(当前时间和时间戳之间的差)为参与者检查一个证书提供了一种风险测度。另一种方法是,利用存储撤销信息的联网服务器,可以用几种在线模式检查一个给定的证书是否有效。

486

3. 利用公钥进行对称密钥交换

公钥密码学克服了对称密钥密码系统中的瓶颈问题。因为在一个公钥密码系统中,在进行安全通信之前,不需要分发密钥。不幸的是,这个优势是有一定代价的,因为现有的公钥密码系统比现有的对称密码系统需要更长的加密和解密时间。因此,在实际中,公钥密码系统常常与对称密码系统结合使用,以克服交换密钥的挑战,以便建立一条对称的密码通信信道。

例如,如果Alice想要建立一条与Bob通信的安全信道,她和Bob可以执行如下步骤集:

(1) Alice计算一个随机数 x ,计算她对 x 的数字签名 S ,并利用Bob的公钥对 (x, S) 进行加密,把所得密文 C 发送给Bob。

(2) Bob利用他的私钥解密 C ,并通过检查签名 S ,验证Alice发送给他的信息。

(3) 然后Bob可以向Alice证明,他确实接收到 x ,他用Alice的公钥对 x 加密,并把它发还给Alice。

从这个观点来看,在对称密码系统中,可以利用数 x 作为密钥。

4. Diffie-Hellman密钥交换

如果Alice和Bob正在利用一种媒介通信,这个媒介是可靠的,但可能不是私有的,则存在另一种模式可用于计算一个密钥,他们可以共享这个密钥,并用于将来的对称加密通信。称这种模式为Diffie-Hellman密钥交换,它由以下步骤组成:

(1) Alice和Bob对一个素数 n 和 Z_n 中的一个生成元 g 达成一致意见(公开)。

(2) Alice选择一个随机数 x ,并向Bob发送 $B = g^x \bmod n$ 。

(3) Bob选择一个随机数 y ,并向Alice发送 $A = g^y \bmod n$ 。

(4) Alice计算 $K = A^x \bmod n$ 。

(5) Bob计算 $K' = B^y \bmod n$ 。

487

显然, $K = K'$,因此Alice和Bob现在通过一个对称密码系统分别利用 K 或 K' 进行通信。

10.4 快速傅里叶变换

在许多密码系统中普遍存在的一个计算瓶颈是大整数乘法和多项式乘法。对于这些对象的乘法,快速傅里叶变换是一种极其有效的算法。首先描述多项式相乘的算法,然后表明如何扩展这个方法,使其应用于大整数相乘的问题。

用系数形式(coefficient form)表示的一个多项式可用系数向量 $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}]$ 描述

如下。

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

这样一个多项式的阶 (degree) 是非零系数 a_i 的最大下标。长为 n 的系数向量可以表示阶至多为 $n-1$ 的多项式。

系数表示法很自然, 这在于它既简单, 又使多项式的几种操作可以快速进行。例如, 给定用系数向量 $\mathbf{b} = [b_0, b_1, \dots, b_{n-1}]$ 描述的第二个多项式如下:

$$q(x) = \sum_{i=0}^{n-1} b_i x^i$$

可以容易地把 $p(x)$ 和 $q(x)$ 的对应项相加, 产生其和

$$p(x) + q(x) = \sum_{i=0}^{n-1} (a_i + b_i) x^i$$

同样, 利用 Horner 法则 (习题 C-1.16), 用于 $p(x)$ 的系数形式可以有效计算 $p(x)$, 如下:

$$p(x) = a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-2} + xa_{n-1}) \dots))$$

因此, 利用系数表示法, 可以用 $O(n)$ 时间计算阶为 $(n-1)$ 的多项式加法和多项式的值。

然而上述系数表示法定义的两个多项式 $p(x)$ 和 $q(x)$ 的相乘并不直观。为了看出难度, 考虑 $p(x)q(x)$:

$$p(x)q(x) = \sum_{i=0}^{2n-2} c_i x^i, \text{ 其中 } c_i = \sum_{j=0}^i a_j b_{i-j}, \quad i = 0, 1, \dots, 2n-2$$

这个方程定义了一个向量 $\mathbf{c} = [c_0, c_1, \dots, c_{2n-1}]$, 称这个向量为向量 \mathbf{a} 和向量 \mathbf{b} 的卷积 (convolution)。由对称性可知, 可把这个卷积看作大小为 $2n$ 的一个向量, 定义 $c_{2n-1} = 0$ 。将 \mathbf{a} 和 \mathbf{b} 的卷积表示为 $\mathbf{a} * \mathbf{b}$ 。如果直接应用卷积的定义, 那么两个多项式 p 和 q 相乘所需时间为 $\Theta(n^2)$ 。

488

快速傅里叶变换 (fast Fourier transform, FFT) 算法进行这两个多项式相乘所需时间为 $O(n \log n)$ 。FFT 对此所做的改进基于一个有趣的观察, 即用 n 个不同的输入值表示阶为 $(n-1)$ 的多项式的另一种方法。由以下定理可知, 这样的表示是唯一的。

定理 10.18 (多项式内插定理) 给定平面上的 n 个点的集合, $S = \{(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_{n-1}, y_{n-1})\}$, 满足 x 互不相同, 那么存在唯一阶为 $(n-1)$ 的多项式 $p(x)$, 且 $p(x_i) = y_i$, 其中 $i = 0, 1, \dots, n-1$ 。

假定我们可以用不同输入集合上的值表示一个多项式, 而不是用多项式的系数表示多项式。这个定理给出了两个多项式 p 和 q 相乘的另一种方法。特别是, 计算 p 和 q 的 $2n$ 个不同的输入值 $x_0, x_1, \dots, x_{2n-1}$ 。并且把 p 和 q 乘积的计算表示为集合:

$$\{(x_0, p(x_0)q(x_0)), (x_1, p(x_1)q(x_1)), \dots, (x_{2n-1}, p(x_{2n-1})q(x_{2n-1}))\}$$

给定 p 和 q 的各 $2n$ 个输入-输出对, 显然这个计算的时间为 $O(n)$ 。

对于 p 和 q 各自的 $2n$ 个输入-输出对, 高效利用这种方法进行 p 和 q 相乘的挑战很快出现。把 Horner 法则应用到 $2n$ 个不同的输入所需时间为 $\Theta(n^2)$, 在渐近意义上这并不比直接利用卷积方法要快。因此, Horner 法则在这里毫无用处。当然, 我们可以完全自由选择多项式的 $2n$ 个输入的一个集合, 即我们有完全的自由来选择易于计算的那些输入。例如, $p(0) = a_0$ 是一种简单情况。但必须选择 $2n$ 个使 p 易于计算的输入集合。幸运的是, 以下讨论的数学概念为我们提供了一个方便的

输入集合, 用这些输入计算多项式的值要比应用 $2n$ 次Horner法则通常更容易一些。

10.4.1 本原单位根

对于 $n \geq 2$, 如果数 ω 满足以下性质:

(1) $\omega^n = 1$, 即 ω 是1的第 n 个根。

(2) 数 $1, \omega, \omega^2, \dots, \omega^{n-1}$ 互不相同。

则称 ω 是第 n 个本原单位根 (primitive n th root of unity)。注意这个定义蕴涵着第 n 个本原单位根存在乘法逆元, $\omega^{-1} = \omega^{n-1}$, 对于

$$\omega^{-1}\omega = \omega^{n-1}\omega = \omega^n = 1$$

489

因此, 我们可以说 ω 的良好定义的负幂形式, 也可以说它的正幂形式。

初看起来, 第 n 个本原单位根的概念似乎像是一个奇怪的定义, 没有几个这样的例子。但它实际上有几个重要的实例。一个重要实例是复数

$$e^{2\pi i/n} = \cos(2\pi/n) + i \sin(2\pi/n)$$

当对复数进行运算时, 它是第 n 个本原单位根, 其中 $i = \sqrt{-1}$ 。

第 n 个本原单位根有许多重要的性质, 包括以下三个性质。

引理10.5 (消去性质) 如果 ω 是第 n 个本原单位根, 那么对于任何整数 $k \neq 0$, 且 $-n < k < n$, 有

$$\sum_{j=0}^{n-1} \omega^{kj} = 0$$

证明 因为 $\omega^k \neq 1$,

$$\sum_{j=0}^{n-1} \omega^{kj} = \frac{(\omega^k)^n - 1}{\omega^k - 1} = \frac{(\omega^n)^k - 1}{\omega^k - 1} = \frac{1^k - 1}{\omega^k - 1} = \frac{1-1}{\omega^k - 1} = 0$$

引理10.6 (归约性质) 如果 ω 是第 $2n$ 个本原单位根, 那么 ω^2 是第 n 个本原单位根。

证明 如果 $1, \omega, \omega^2, \dots, \omega^{2n-1}$ 互不相同, 那么 $1, \omega^2, (\omega^2)^2, \dots, (\omega^2)^{n-1}$ 也互不相同。■

引理10.7 (反射性质) 如果 ω 是第 n 个本原单位根, 且 n 是偶数, 那么

$$\omega^{n/2} = -1$$

证明 对于 $k = n/2$, 由消去性可知,

$$\begin{aligned} 0 &= \sum_{j=0}^{n-1} \omega^{(n/2)j} \\ &= \omega^0 + \omega^{n/2} + \omega^n + \omega^{3n/2} + \dots + \omega^{(n/2)(n-2)} + \omega^{(n/2)(n-1)} \\ &= \omega^0 + \omega^{n/2} + \omega^0 + \omega^{n/2} + \dots + \omega^0 + \omega^{n/2} \\ &= (n/2)(1 + \omega^{n/2}) \end{aligned}$$

因此, $0 = 1 + \omega^{n/2}$ 。■

由此得名的关于反射性的一个有趣的推论是, 如果 ω 是第 n 个本原单位根, 且 $n \geq 2$ 是偶数, 那么

$$\omega^{k+n/2} = -\omega^k$$

490

10.4.2 离散傅里叶变换

对于经过仔细选择的输入值集合, 现在回到通过系数向量 \mathbf{a} 计算多项式

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

值的问题。称本节中讨论的技术为离散傅里叶变换 (discrete Fourier transform, DFT), 用于计算 $p(x)$ 在 $\omega^0, \omega^1, \omega^2, \dots, \omega^{n-1}$ 处的第 n 个本原单位根。诚然, 这正好给出 n 个输入-输出对, 但是可通过设 $a_i = 0$ (其中 $n \leq i \leq 2n-1$), 在 p 的系数表示中“填补”0。这种填补可把 p 看作阶为 $(2n-1)$ 的多项式。这反过来使我们可以利用第 $2n$ 个本原单位根作为 p 的DFT的输入。如果我们需要 p 的更多输入-输出值, 则假设已用所需的多个0填补在 p 的系数向量中。

形式上, 系数向量 \mathbf{a} 表示的多项式 p 的离散傅里叶变换被定义为向量 \mathbf{y} 的值

$$y_j = p(\omega^j)$$

其中 ω 是第 n 个本原单位根, 即

$$y_j = \sum_{i=0}^{n-1} a_i \omega^{ij}$$

可把值 y_j 看作向量 \mathbf{y} , 把向量 \mathbf{a} 看作列向量, 用矩阵表示为

$$\mathbf{y} = F\mathbf{a}$$

其中 F 是满足 $F[i, j] = \omega^{ij}$ 的 $n \times n$ 矩阵。

1. 离散傅里叶逆变换

有趣的是, 矩阵 F 存在逆矩阵 F^{-1} , 满足对于所有 \mathbf{a} , $F^{-1}(f(\mathbf{a})) = \mathbf{a}$ 。利用矩阵 F^{-1} 可以定义离散傅里叶逆变换 (inverse discrete Fourier transform)。如果给定阶为 $(n-1)$ 的多项式 $p(x)$ 在 $\omega^0, \omega^1, \omega^2, \dots, \omega^{n-1}$ 处的第 n 个本原单位根的值组成的向量 \mathbf{y} , 那么通过计算

$$\mathbf{a} = F^{-1}\mathbf{y}$$

可以恢复 p 的系数向量。此外, 矩阵 F^{-1} 具有简单形式, 这在于 $F^{-1}[i, j] = \omega^{-ij}/n$ 。因此, 可把系数 a_i 恢复为

$$a_i = \sum_{j=0}^{n-1} y_j \omega^{-ij} / n$$

以下引理证实了这个声明, 它也是为什么称 F 和 F^{-1} 为“变换”的基础。

491

引理10.8 对于任意向量 \mathbf{a} , $F^{-1} \cdot F\mathbf{a} = \mathbf{a}$ 。

证明 设 $A = F^{-1} \cdot F$ 。只要证明如果 $i = j$, 则 $A[i, j] = 1$; 如果 $i \neq j$, 则 $A[i, j] = 0$, 即 $A = I$, 其中 I 是恒等矩阵 (identity matrix)。由 F^{-1} 、 F 和矩阵相乘的定义, 可知

$$A[i, j] = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-ik} \omega^{kj}$$

如果 $i = j$, 那么这个方程可化简为

$$A[i, i] = \frac{1}{n} \sum_{k=0}^{n-1} \omega^0 = \frac{1}{n} \cdot n = 1$$

因而, 考虑 $i \neq j$ 时的情况, 设 $m = j - i$ 。那么 A 的第 ij 个元素可写为

$$A[i, j] = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{mk}$$

其中 $-n < m < n$, 且 $m \neq 0$ 。由第 n 个本原单位根的消去性质, 将上述方程的右边化简为0; 因此, 对于 $i \neq j$, 有

$$A[i, j] = 0$$

给定DFT和逆DFT, 现在可以定义两个多项式 p 和 q 相乘的方法。

2. 卷积定理

为了使用离散傅里叶变换及其逆变换计算两个系数向量 \mathbf{a} 和 \mathbf{b} 的卷积, 可应用以下步骤, 图10-1说明了其原理图。

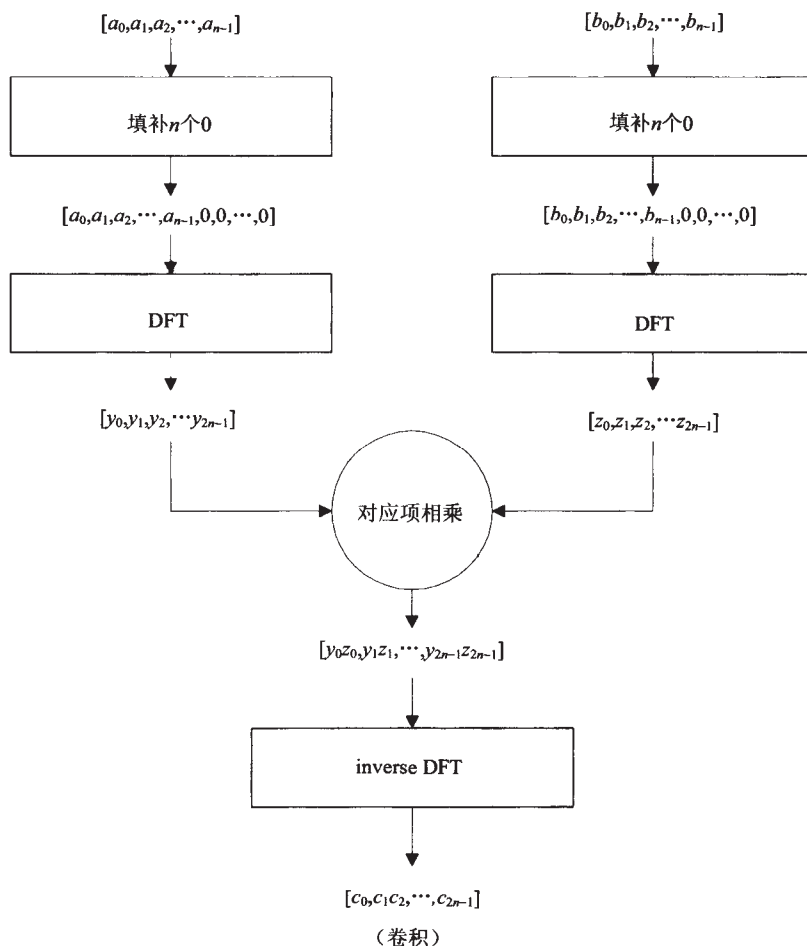


图10-1 卷积定理的说明, 计算 $\mathbf{c} = \mathbf{a} * \mathbf{b}$

(1) 在 \mathbf{a} 和 \mathbf{b} 中填补 n 个0, 把它们看作列向量, 定义为:

$$\mathbf{a}' = [a_0, a_1, \dots, a_{n-1}, 0, 0, \dots, 0]^T$$

$$\mathbf{b}' = [b_0, b_1, \dots, b_{n-1}, 0, 0, \dots, 0]^T$$

- (2) 计算离散傅里叶变换 $y = Fa'$ 和 $z = Fb'$ 。
 (3) 把向量 y 和 z 对应项相乘, 定义简单乘积 $y \cdot z = Fa' \cdot Fb'$, 其中

$$(y \cdot z)[i] = (Fa' \cdot Fb')[i] = Fa'[i] \cdot Fb'[i] = y_i \cdot z_i$$

其中 $i = 1, 2, \dots, 2n-1$ 。

- (4) 计算这个简单乘积的离散傅里叶逆变换, 即计算 $c = F^{-1}(Fa' \cdot Fb')$ 。

上述方法可行的原因在于以下定理。

492
493

定理10.19 (卷积定理) 给定两个长为 n 的向量 a 和 b , 分别填补0使它们变成长为 $2n$ 的向量 a' 和 b' 。那么 $a * b = F^{-1}(Fa' \cdot Fb')$ 。

证明 要证明 $F(a * b) = Fa' \cdot Fb'$ 。因而考虑 $A = Fa' \cdot Fb'$ 。由于 a' 和 b' 的后半部分是用0填补的, 所以有

$$\begin{aligned} A[i] &= \left(\sum_{j=0}^{n-1} a_j \omega^{ij} \right) \cdot \left(\sum_{k=0}^{n-1} b_k \omega^{ik} \right) \\ &= \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k \omega^{i(j+k)} \end{aligned}$$

其中 $i = 0, 1, \dots, 2n-1$ 。下面考虑 $B = f(a * b)$ 。由卷积和DFT的定义可知:

$$B[i] = \sum_{l=0}^{2n-1} \sum_{j=0}^{2n-1} a_j b_{l-j} \omega^{il}$$

用 k 代换 $l-j$, 并改变求和顺序, 得

$$B[i] = \sum_{j=0}^{2n-1} \sum_{k=-j}^{2n-1-j} a_j b_k \omega^{i(j+k)}$$

因为 $k < 0$ 时, b_k 未定义, 可以在 $k=0$ 时开始执行上述第二个求和式。此外, 对于 $j > n-1$, 由于 $a_j = 0$, 可把第一个求和式的上限降至 $n-1$ 。但一旦做出这种代换, 注意到上面第二个求和式的上限总是至少为 n 。因此, 对于 $k > n-1$, 由于 $b_k = 0$, 可以把第二个求和式的上限降至 $n-1$, 于是

$$B[i] = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k \omega^{i(j+k)}$$

定理证毕。 ■

现在有了计算两个多项式相乘的方法, 它涉及计算两个DFT、进行一个简单的线性时间的对应项相乘和计算一个逆DFT。因此, 如果可以找到计算DFT及其逆DFT的快速方法, 那么就得到两个多项式相乘的快速算法。下一节将描述这样一个快速算法, 称之为“快速傅里叶变换”。

494

10.4.3 快速傅里叶变换算法

快速傅里叶变换 (Fast Fourier Transform, FFT) 算法计算长为 n 的向量的离散傅里叶变换 (DFT) 的时间为 $O(n \log n)$ 。在FFT算法中, 利用分治法对多项式求值。观察可见, 如果 n 是偶数, 可以把阶为 $(n-1)$ 的多项式:

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$$

划分成两个阶为 $(n/2-1)$ 的多项式:

$$p^{\text{even}}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}$$

$$p^{\text{odd}}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}$$

并注意到, 利用方程

$$p(x) = p^{\text{even}}(x^2) + xp^{\text{odd}}(x^2)$$

可以把这两个多项式组合成 p 。DFT计算 $p(x)$ 在 $\omega^0, \omega^1, \omega^2, \dots, \omega^{n-1}$ 处的第 n 个本原单位根。注意由归约性质可知, 值 $(\omega^2)^0, \omega^2, (\omega^2)^2, (\omega^2)^3, \dots, (\omega^2)^{n-1}$ 是第 $(n/2)$ 个本原单位根。因此, 可以计算这些值上的 $p^{\text{even}}(x)$ 和 $p^{\text{odd}}(x)$ 。并在计算 $p(x)$ 中重用那些相同的计算。这个观察用在算法10-10(FFT)中, 算法输入为长为 n 的系数向量 \mathbf{a} 和第 n 个本原单位根 ω 。为简明起见, 假设 n 是2的幂。

算法10-10 递归FFT算法

算法 FFT(\mathbf{a}, ω)

输入: 长为 n 的系数向量 $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}]$ 和第 n 个本原单位根 ω , 其中 n 是2的幂。

输出: \mathbf{a} 的多项式在第 n 个本原单位根的值组成的向量 \mathbf{y}

if $n = 1$ then

return $\mathbf{y} = \mathbf{a}$

$x \leftarrow \omega^0$ { x 存储 ω 的幂, 因此起初 $x = 1$ }

{划分步骤, 分割偶数下标和奇数下标}

$\mathbf{a}^{\text{even}} \leftarrow [a_0, a_2, a_4, \dots, a_{n-2}]$

$\mathbf{a}^{\text{odd}} \leftarrow [a_1, a_3, a_5, \dots, a_{n-1}]$

{递归调用, 由归约性质可知, ω^2 为第 $(n/2)$ 个本原单位根。}

$\mathbf{y}^{\text{even}} \leftarrow \text{FFT}(\mathbf{a}^{\text{even}}, \omega^2)$

$\mathbf{y}^{\text{odd}} \leftarrow \text{FFT}(\mathbf{a}^{\text{odd}}, \omega^2)$

{组合步骤, 利用 $x = \omega$ }

for $i \leftarrow 0$ to $n/2 - 1$ do

$y_i \leftarrow y_i^{\text{even}} + x \cdot y_i^{\text{odd}}$

$y_{i+n/2} \leftarrow y_i^{\text{even}} - x \cdot y_i^{\text{odd}}$ {利用反射性质}

$x \leftarrow x \cdot \omega$

return \mathbf{y}

495

1. FFT算法的正确性

算法10-10中关于FFT算法的伪代码描述简单得令人困惑。因此我们解释一下它如何能正确工作。首先, 注意递归基础, 当 $n=1$ 时, 正确返回一个元素 $y_0 = a_0$ 的向量 \mathbf{y} , 此时, 它是多项式 $p(x)$ 中的最高阶项, 且只有这一项。

在一般情况下, 当 $n \geq 2$ 时, 把 \mathbf{a} 分解成 \mathbf{a}^{even} 偶实例和 \mathbf{a}^{odd} 奇实例, 并利用 ω^2 作为第 $(n/2)$ 个本原单位根, 递归调用FFT。正如已经提到的那样, 第 n 个本原单位根的归约性质使得可以这样利用 ω^2 。因此, 可以导出假设

$$y_i^{\text{even}} = p^{\text{even}}(\omega^{2i})$$

$$y_i^{\text{odd}} = p^{\text{odd}}(\omega^{2i})$$

考虑组合递归调用所得值的for循环, 注意在循环的第 i 次迭代中, $x = \omega^i$ 。因此, 当执行赋值语句时,

$$y_i \leftarrow y_i^{\text{even}} + x y_i^{\text{odd}}$$

则有集合

$$\begin{aligned} y_i &= p^{\text{even}}((\omega^2)^i) + \omega^i \cdot p^{\text{odd}}((\omega^2)^i) \\ &= p^{\text{even}}((\omega^j)^2) + \omega^i \cdot p^{\text{odd}}((\omega^j)^2) \\ &= p(\omega^j) \end{aligned}$$

且对于每个 $i = 0, 1, \dots, n/2 - 1$ 的下标进行。类似地, 当执行赋值语句时,

$$y_{i+n/2} \leftarrow y_i^{\text{even}} - x y_i^{\text{odd}}$$

则有集合

$$y_{i+n/2} = p^{\text{even}}((\omega^2)^i) - \omega^i \cdot p^{\text{odd}}((\omega^2)^i)$$

由于 ω^2 为第 $(n/2)$ 个本原单位根, $(\omega^2)^{n/2} = 1$ 。此外, 由于 ω 自身是第 n 个本原单位根, 由反射性质可知

$$\omega^{j+n/2} = -\omega^j$$

因此, 可以将上述关于 $y_{i+n/2}$ 的等式重写为

$$\begin{aligned} y_{i+n/2} &= p^{\text{even}}((\omega^2)^{i+(n/2)}) - \omega^i \cdot p^{\text{odd}}((\omega^2)^{i+(n/2)}) \\ &= p^{\text{even}}((\omega^{j+(n/2)})^2) + \omega^{j+n/2} \cdot p^{\text{odd}}((\omega^{j+(n/2)})^2) \\ &= p(\omega^{j+n/2}) \end{aligned}$$

这个式子对于每个 $i = 0, 1, \dots, n/2 - 1$ 均成立。因此, FFT算法返回的向量 y 将 $p(x)$ 的值存储在每个第 n 个本原单位根中。

496

2. 分析FFT算法

FFT算法遵循分治法范型, 把规模为 n 的问题分解成规模为 $n/2$ 的两个子问题, 并递归求解子问题。假设算法中每次进行的算术运算需要 $O(1)$ 时间。对于划分步骤以及归并递归解的组合步骤, 其中每个步骤都需要 $O(n)$ 时间。因此, 利用递归方程

$$T(n) = 2T(n/2) + bn$$

表征FFT算法的运行时间 $T(n)$, 其中常数 $b > 0$ 。由主定理(5.6)可知, $T(n)$ 为 $O(n \log n)$ 。因此, 可以将讨论概括如下:

定理10.20 给定定义多项式 $p(x)$ 的长为 n 的系数向量 a , 以及第 n 个本原单位根 ω , FFT算法计算 $p(x)$ 在每一个第 n 个本原单位根 ω^j 处的值, 所需时间为 $O(n \log n)$, 其中 $i = 0, 1, \dots, n-1$ 。

还有一个逆FFT算法, 它用 $O(n \log n)$ 时间计算逆DFT。这个算法的细节类似于FFT算法的那些细节, 把它留作习题(R-10.14)。在两个多项式 $p(x)$ 和 $q(x)$ 相乘的方法中, 把这两个算法组合起来, 所得算法计算这个乘积的时间为 $O(n \log n)$ 。

顺便说一句, 利用FFT算法和它的逆算法计算两个多项式相乘的方法可以扩展到计算两个大整数相乘的问题。下一节中讨论这个方法。

10.4.4 大整数相乘

重新回顾5.2.2节研究的问题, 即给定两个大整数 I 和 J , 每个至多64位, 我们对计算 $I \cdot J$ 感兴趣。这一节里所描述方法的主要思想是利用FFT算法计算这个乘积。当然, 利用FFT这样的算法设计方法面临的主要挑战是定义可以引出本原单位根的整数运算(例如, 参见习题C-10.8)。

这里描述的算法假设可把 I 和 J 分解为每个长为 $O(\log N)$ 位的字,使得可以利用计算机模型中的内置操作在常数时间完成对每个字的运算。称这个模型为字-RAM模型(word-RAM model)。在这个模型中, I 和 J 相乘是合理的。因为它假设字不是太长,也不是太短。例如,只需要 $\lceil \log N \rceil$ 位表示数 N 自身,可以用一个字表示 N 。在这种情况下,对于选择的可用一个内存字表示的合适素数 p ,FFT使用的数制系统执行所有模 p 的算术运算。为简化描述,还假设 n 是2的幂。

497

选择的特殊素数 p 是一个形如 $2^a b + 1$ 的素数,其中 a 和 b 是两个整数, a 靠近字-RAM模型的字长,仍然使 p 可以表示成一个字。注意,假设 n 为2的幂,这样选择的 p 蕴涵着 $p = cn + 1$,其中 c 为某个整数。数论的基本定理阐明,范围在 $[1, X]$ 内且形如 $2^a b + 1$ 的随机奇数是素数的概率为 $\Omega(1/\log X)$ 。对于给定的 a ,大约有 $X/2^a$ 个这样的数。因此,可以用 $\Theta(\log N)$ 位表示 p 。给定素数 p ,我们的意图是在FFT模 p 中执行所有算术运算。

给定这样的素数 p ,选择基大小(base size), $m < \log p$,使我们可以把 I 和 J 分别看成长为 n 的向量 \mathbf{a} 和 \mathbf{b} ,将它们表示成以 2^m 为基的形式。而且,我们还假设根据需要可以用0扩展 I 和 J ,使 \mathbf{a} 和 \mathbf{b} 的长度都达到 n ,且至少高阶项后半部分为0,即可以把 I 和 J 写为

$$I = \sum_{i=0}^{n-1} a_i 2^{mi}$$

$$J = \sum_{i=0}^{n-1} b_i 2^{mi}$$

使得 a_i 和 b_i 的至多前 $n/2$ 项是非零的。因此,可把乘积 $K = I \cdot J$ 表示为

$$K = \sum_{i=0}^{n-1} c_i 2^{mi}$$

选择的 m 值应该至少为 $\log N$,其中 $n < N \leq 2^m$ 。同时希望计算 I 和 J 中由那些小于 p 的系数组成的向量的卷积。定义 \mathbf{a} 和 \mathbf{b} 的卷积中的每一项为至多 n 项的和。每一项是一个 a_i 和一个 b_j 的乘积, I 和 J 卷积中的每个系数可用 $2m + \log n$ 位表示。于是选择 $m = \lfloor (\log p)/3 \rfloor$,假设字-RAM模型中 p 可用一个字表示。例如,在以下条件下,标准计算机上所用的这个算法可以选择如下 p 值:

- 如果选择 $p = 15 \cdot 2^{27} + 1 = 2\,013\,265\,921$ 作为素数模数,那么可在一个32位字(因为 p 使用了31位)中进行所有算术运算。而且,由于 $p > 2^{30}$,在这种情况下,可以把那些每个字10位,其表示达到 n ($\leq 2^{10}$)个字的数(即这些数和 $2^{10 \cdot 240} - 1$ 一样大)相乘。

一旦确定素数 $p = cn + 1$,其中 $c \geq 1$ 为合理小的整数,就找到一个整数 x ,它是群 Z_p^* 的生成元(见10.1节)。也就是说,找到 x ,满足 $x^i \bmod p$ 是不同的值,其中 $i = 0, 1, \dots, p-1$ 。给定这样一个生成元 x ,那么可以利用 $\omega = x^c \bmod p$ 作为第 n 个本原单位根(假设所有加法和乘法都要执行模 p)。也就是说, $(x^c)^i \bmod p$ 中的每个数互不相同,其中 $i = 0, 1, \dots, n-1$,但由费马小定理(定理10.6)可知,

498

$$\begin{aligned} (x^c)^n \bmod p &= x^{cn} \bmod p \\ &= x^{p-1} \bmod p \\ &= 1 \end{aligned}$$

为了计算 N 位大整数 I 和 J 的乘积,回忆一下可把 I 和 J 分别看作扩展的长为 n 个字的向量 \mathbf{a} 和 \mathbf{b} ,其中每个字都有 m 位(至少 $n/2$ 个高阶字都为0)。注意,因为 $m = \lfloor \log p/2 \rfloor$,则有 $2^m < p$ 。因此, \mathbf{a} 和 \mathbf{b} 中的每一项已经归约到模 p 上,而不需做任何额外的工作。

为了计算 $K = I \cdot J$,然后应用卷积定理,利用FFT算法和逆FFT算法计算 \mathbf{a} 和 \mathbf{b} 的卷积 \mathbf{c} 。在这个计算中,利用 ω (如上述定义)作为第 n 个本原单位根,进行所有模 p 的算术运算(包括 \mathbf{a} 和 \mathbf{b} 变换后的对应项的乘积)。

卷积 \mathbf{c} 中的每一项都小于 p ，但为构造 $K = I \cdot J$ 的一种表示，实际上需要把每一项恰好表示成 m 位。因此，得到卷积 \mathbf{c} 之后，必须计算乘积 K ，如下：

$$K = \sum_{i=0}^{n-1} c_i 2^{mi}$$

这个最后的计算并不像它看起来的那么难。因为乘以2的幂在二进制中正好是一个移位操作。而且 p 是 $O(2^{m+1})$ 。上述求和只是涉及从一项向下一项的进位传播。因此，把 K 的二进制表示最终构造成为每个具有 m 位的字组成的向量所需时间为 $O(n)$ 。因为应用上述的卷积定理需要 $O(n \log n)$ 的时间。由此可得如下定理。

定理10.21 给定两个 N 位的整数 I 和 J ，则计算 $K = I \cdot J$ 的乘积所需时间为 $O(n)$ ，假设涉及大小为 $O(\log N)$ 的字的算术操作的完成时间为常数时间。

证明 选择数 n 为 $O(N / \log N)$ 。假设涉及大小为 $O(\log N)$ 的字的算术操作的完成时间为常数时间，因此 $O(n \log n)$ 的运行时间为 $O(N)$ 。 ■

在某些情况下，不能假设涉及大小为 $O(\log N)$ 的字的算术操作的完成时间为常数时间，而是假设每个位操作必须花费常数时间。在这个模型中，仍然可能利用FFT进行两个 N 位的大整数相乘。但是细节更复杂，且运行时间增加到 $O(N \log N \log \log N)$ 。

在10.5节中，研究与FFT算法相关的一些实现问题，包括在实际中它如何有效执行的实验性分析。

499

10.5 Java 示例：FFT

在这一节里讨论许多与FFT算法有关的有趣实现问题。首先描述一个大整数类，它利用递归FFT算法进行乘法计算，如同上述算法10-10的伪代码中描述的那样。代码段10-1中给出了这个类的声明、以及它的重要实例变量和常量。这个声明中的重要细节包括定义第 n 个本原单位根 Ω 。选择 31^{15} 是由于 31 是 Z_p^* 的关于素数 $15 \cdot 2^{27} + 1$ 的一个生成元。我们之所以知道 31 是这个 Z_p^* 的一个生成元，是由于数论中的一个定理。这个定理指出当且仅当对于 $\phi(p)$ 的任何素因子 q ， $x^{\phi(p)/q} \bmod p$ 不等于 1 时，整数 x 才是 Z_p^* 的一个生成元。其中 $\phi(p)$ 是10.1节定义的欧拉 ϕ 函数。对于特殊 p ， $\phi(p) = 15 \cdot 2^{27}$ ；因此，要发现 31 是 Z_p^* 的一个生成元，需要考虑的唯一素因子 q 为 2 、 3 和 5 。

代码段10-1 支持利用FFT算法计算乘法的大整数类的声明及其实例变量

```
import java.lang.*;
import java.math.*;
import java.util.*;

public class BigInt {
    protected int signum=0;           // neg = -1, 0 = 0, pos = 1
    protected int[] mag;              // magnitude in little-endian format
    public final static int MAXN=134217728; // Maximum value for n
    public final static int ENTRYSIZE=10; // Bits per entry in mag
    protected final static long P=2013265921; // The prime  $15 \cdot 2^{27} + 1$ 
    protected final static int OMEGA=440564289; // Root of unity  $31^{\{15\}} \bmod P$ 
    protected final static int TWOINV=1006632961; //  $2^{-1} \bmod P$ 
```

1. 递归FFT实现

代码段10-2中给出了大整数类中的相乘方法BigInt。代码段10-3中给出了递归FFT算法的实

现。注意我们利用了一个变量`prod`存储后面两个表达式要使用的乘积。对这个公共子表达式进行因式分解，可以避免进行这个重复计算两次。此外，注意把所有模运算的执行作为`long`型操作。这个要求是由于`P`利用了31位这一事实，因此我们进行的加法和减法运算可能使标准整数大小溢出。作为替代我们把所有算术运算作为`long`型操作来执行，然后把结果存储回`int`型。

500

代码段10-2 支持利用递归FFT算法计算乘法的大整数类的乘法方法

```
public BigInt multiply(BigInt val) {
    int n = makePowerOfTwo(Math.max(mag.length, val.mag.length))*2;
    int signResult = signum * val.signum;
    int[] A = padWithZeros(mag, n); // copies mag into A padded w/ 0's
    int[] B = padWithZeros(val.mag, n); // copies val.mag into B padded w/ 0's
    int[] root = rootsOfUnity(n); // creates all n roots of unity
    int[] C = new int[n]; // result array for A*B
    int[] AF = new int[n]; // result array for FFT of A
    int[] BF = new int[n]; // result array for FFT of B
    FFT(A, root, n, 0, AF);
    FFT(B, root, n, 0, BF);
    for (int i=0; i<n; i++)
        AF[i] = (int)(((long)AF[i]*(long)BF[i]) % P); // Component multiply
    reverseRoots(root); // Reverse roots to create inverse roots
    inverseFFT(AF, root, n, 0, C); // Leaves inverse FFT result in C
    propagateCarries(C); // Convert C to right no. bits per entry
    return new BigInt(signResult, C);
}
```

代码段10-3 FFT算法的递归实现

```
public static void FFT(int[] A, int[] root, int n, int base, int[] Y) {
    int prod;
    if (n==1) {
        Y[base] = A[base];
        return;
    }
    inverseShuffle(A, n, base); // inverse shuffle to separate evens and odds
    FFT(A, root, n/2, base, Y); // results in Y[base] to Y[base+n/2-1]
    FFT(A, root, n/2, base+n/2, Y); // results in Y[base+n/2] to Y[base+n-1]
    int j = A.length/n;
    for (int i=0; i<n/2; i++) {
        prod = (int)(((long)root[i*j]*Y[base+n/2+i]) % P);
        Y[base+n/2+i] = (int)(((long)Y[base+i] + P - prod) % P);
        Y[base+i] = (int)(((long)Y[base+i] + prod) % P);
    }
}

public static void inverseFFT(int[] A, int[] root, int n, int base, int[] Y) {
    int inverseN = modInverse(n); // n^{-1}
    FFT(A, root, n, base, Y);
    for (int i=0; i<n; i++)
        Y[i] = (int)(((long)Y[i]*inverseN) % P);
}
```

501

2. 避免数组重复分配

递归FFT算法的伪代码要求为几个新数组分配空间，这些数组是 a^{even} 、 a^{odd} 、 y^{even} 、 y^{odd} 和 y 。可以证明每次调用中为这些数组分配空间是一项昂贵的额外工作。如果能够避免做这项工作，节

省这个额外分配数组的时间，就能大大改进执行FFT算法的 $O(n \log n)$ （或 $O(N)$ ）时间性能中的常数因子。

幸运的是，FFT的结构使我们可以避免这个重复分配。我们不是分配许多个数组，而可以只用一个数组 A 用于输入系数，一个数组 Y 用于结果。这种用法的主要思想是可以把数组 A 和 Y 划分成子数组，每个子数组关联一个不同的递归调用。只利用两个变量 $base$ 和 n ，分别用于表示子数组的基址和子数组的大小。因此，我们可以避免每次递归调用中与分配许多小数组关联的开销。

3. 逆混洗

在确定在FFT递归调用过程中不分配新的数组空间之后，必须处理如下事实：即FFT涉及在输入数组的偶数下标和奇数下标上执行独立的计算。在算法10-10的伪代码中，利用新数组 a^{even} 和 a^{odd} ，但现在必须利用子数组 A 表示这些向量。这个内存管理问题的解决方案是取 A 中当前 n 个单元的子数组，把它划分成大小为 $n/2$ 的两个子数组。一个子数组的基下标与 A 一样，另一个子数组的基下标为 $base + n/2$ 。把 A 中具有偶数下标的元素移到 A 中低半部分，把 A 中具有奇数下标的元素移到 A 中高半部分。在进行过程中，定义一个称为逆混洗（inverse shuffle）的有趣排列，由于这个排列与其逆排列相似而得此名。把数组 A 分成两半，就像一堆牌，完全进行混洗（见图10-2）。

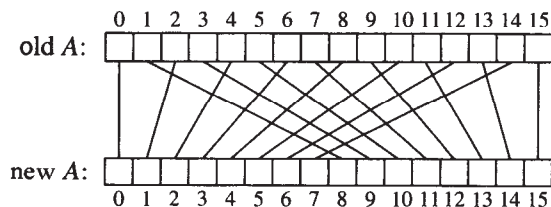


图10-2 逆混洗排列的说明

502

4. 预先计算本原单位根和其他优化措施

在FFT算法实现中，还可利用许多额外的优化方法。在代码段10-4中，显示了实现中的一些重要支持方法，包括 n^{-1} 逆元的计算、逆混洗计算、所有第 n 个本原单位根的预先计算和计算模 p 的卷积之后的进位传播。

代码段10-4 递归FFT的支持方法

```
protected static int modInverse(int n) { // assumes n is power of two
    int result = 1;
    for (long twoPower = 1; twoPower < n; twoPower *= 2)
        result = (int)((long)result*TWOINV % P);
    return result;
}

protected static void inverseShuffle(int[] A, int n, int base) {
    int shift;
    int[] sp = new int[n];
    for (int i=0; i<n/2; i++) { // Unshuffle A into the scratch space
        shift = base + 2*i;
        sp[i] = A[shift]; // an even index
        sp[i+n/2] = A[shift+1]; // an odd index
    }
    for (int i=0; i<n; i++)
        A[base+i] = sp[i]; // copy back to A
}

protected static int[] rootsOfUnity(int n) { //assumes n is power of 2
    int t = MAXN;
```

```

int nthroot = OMEGA;
for (int t = MAXN; t > n; t /= 2) // Find prim. nth root of unity
    nthroot = (int)((long)nthroot*nthroot) % P);
int[] roots = new int[n];
int r = 1; // r will run through all nth roots of unity
for (int i=0; i<n; i++) {
    roots[i] = r;
    r = (int)((long)r*nthroot) % P);
}
return roots;
}
protected static void propagateCarries(int[] A) {
    int i, carry;
    carry = 0;
    for (i=0; i<A.length; i++) {
        A[i] = A[i] + carry;
        carry = A[i] >>> ENTRYSIZE;
        A[i] = A[i] - (carry << ENTRYSIZE);
    }
}

```

503

5. 迭代FFT实现

还可以对FFT算法进行额外的时间上的改进。用它的迭代版本替代递归版本。迭代FFT是另一个大整数类（称为FastInt）的一部分。这个类的乘法方法在代码段10-5中显示。

代码段10-5 迭代FFT的乘法方法

```

public FastInt multiply(FastInt val) {
    int n = makePowerOfTwo(Math.max(mag.length, val.mag.length))*2;
    logN = logBaseTwo(n); // Log of n base 2
    reverse = reverseArray(n, logN); // initialize reversal lookup table
    int signResult = signum * val.signum;
    int[] A = padWithZeros(mag, n); // copies mag into A padded w/ 0's
    int[] B = padWithZeros(val.mag, n); // copies val.mag into B padded w/ 0's
    int[] root = rootsOfUnity(n); // creates all n roots of unity
    FFT(A, root, n); // Leaves FFT result in A
    FFT(B, root, n); // Leaves FFT result in B
    for (int i=0; i<n; i++)
        A[i] = (int) (((long)A[i]*B[i]) % P); // Component-wise multiply
    reverseRoots(root); // Reverse roots to create inverse roots
    inverseFFT(A, root, n); // Leaves inverse FFT result in A
    propagateCarries(A); // Convert A to right no. of bits/entry
    return new FastInt(signResult, A);
}

```

6. 原位计算FFT

由代码段10-5已经可以看出FFT迭代版本和递归版本的乘法方法的差别，即现在可以执行原位FFT。也就是数组A可同时用于输入值和输出值。这种方法也节省了在输出数组和输入数组之间的复制工作。此外，计算n的以2为底的对数，并把结果存储在一个静态变量中，这是因为在调用迭代FFT算法时，会反复利用这个对数值。

7. 避免递归

在FFT算法的原位版本中，避免递归面临的主要挑战是，在输入数组A中必须给出进行所有逆混洗的一种方式。我们不是在每次迭代时执行每个逆混洗过程，而是预先执行所有逆混洗过程。假设输入数组大小n是2的幂。

为了理解通过反复和递归逆混洗操作所得排列的真正作用,考虑在每次递归调用前后逆混洗是如何移动数据的。在第一次递归调用中,在整个数组 A 上执行一次逆混洗过程。注意这个排列是如何在 A 中下标的位级操作的。把最低有效位为0的地址中的所有元素放到 A 的下半部分。同样,把最低有效位为1的地址中的所有元素放到 A 的上半部分。也就是说,如果一个元素开始于地址中的最低有效位 b 处,那么它结束于地址中的最高有效位 b 处。地址中的最低有效位决定一个元素存在于 A 中的哪半部分。在下一级递归调用中,在 A 中每一半上重复逆混洗过程。再次观察 $b=0,1$ 的位级,这些递归逆混洗起初取第二个最低有效位为 b 的地址中的元素,并把它们移动到第二个最高有效位为 b 的地址中。同样,对于 $b=0,1$,第 i 级的递归调用起初取第 i 个最低有效位为 b 的地址中的元素,并把它们移动到第 i 个最高有效位为 b 的地址中。因此,如果元素开始于二进制表示为 $[b_{l-1}\cdots b_2b_1b_0]$ 的地址,那么它在二进制表示为 $[b_0b_1b_2\cdots b_{l-1}]$ 的地址处结束,其中 $l=\log_2 n$ 。也就是说,只需将 A 中元素移到 A 中与其开始地址颠倒的位置,就可以进行所有逆混洗过程。为了进行这种排列,在multiply方法中构造一个排列数组reverse,然后在FFT方法内调用的bitReversal方法中利用这个数组,按照这种排列方式对输入数组 A 中的元素进行排列。所得FFT算法的迭代版本如代码段10-6所示。

504

代码段10-6 FFT算法的迭代实现

```
public static void FFT(int[] A, int[] root, int n) {
    int prod, term, index; // Values for common subexpressions
    int subSize = 1; // Subproblem size
    bitReverse(A, logN); // Permute A by bit reversal table
    for (int lev=1; lev<=logN; lev++) {
        subSize *= 2; // Double the subproblem size.
        for (int base=0; base<n-1; base += subSize) { // Iterate subproblems
            int j = subSize/2;
            int rootIndex = A.length/subSize;
            for (int i=0; i<j; i++) {
                index = base + i;
                prod = (int) (((long)root[i*rootIndex]*A[index+j]) % P);
                term = A[index];
                A[index+j] = (int) (((long)term + P - prod) % P);
                A[index] = (int) (((long)term + prod) % P);
            }
        }
    }
}

public static void inverseFFT(int[] A, int[] root, int n) {
    int inverseN = modInverse(n); // n^{-1}
    FFT(A, root, n);
    for (int i=0; i<n; i++)
        A[i] = (int) (((long)A[i]*inverseN) % P);
}
```

505

代码段10-7中显示了迭代FFT算法中使用的其他一些支持方法。所有这些支持方法都与计算reverse排列表有关,并且利用它在 A 上进行位-逆排列。

代码段10-7 FFT算法迭代实现的支持方法。其他支持方法如同那些在递归实现中的方法

```
protected static void bitReverse(int[] A, int logN) {
    int[] temp = new int[A.length];
    for (int i=0; i<A.length; i++)
        temp[reverse[i]] = A[i];
}
```

```

    for (int i=0; i<A.length; i++)
        A[i] = temp[i];
}
protected static int[] reverseArray(int n, int logN) {
    int[] result = new int[n];
    for (int i=0; i<n; i++)
        result[i] = reverse(i, logN);
    return result;
}
protected static int reverse(int N, int logN) {
    int bit=0;
    int result=0;
    for (int i=0; i<logN; i++) {
        bit = N & 1;
        result = (result << 1) + bit;
        N = N >>> 1;
    }
    return result;
}

```

8. 实验性结果

当然，利用FFT算法进行大整数相乘的目标是在 $O(n \log n)$ 时间内执行这个操作。而对于表示成 n 个字向量的整数，中学课本里所讲授的标准乘法算法时间为 $O(n^2)$ 。此外，我们设计了FFT算法的一种迭代版本，改进了这个乘法方法运行时间中的常数因子。为实际测试这些目标，我们设计了一个简单的实验。

在这个实验中，随机产生10个大整数，其中每个整数由 2^s 个字组成，每个字15位，其中 $s = 7, 8, \dots, 16$ ，使一对连续的数相乘，得到每个 s 值的9个乘积。记录执行这些乘积的运行时间，并与用Java BigInteger类的标准实现中同位数表示的整数相乘时间进行比较。实验结果显示在图10-3中。运行时间以毫秒计。实验环境为Sun Ultra5，软件为用于JDK 1.2的Sun Java虚拟机，主频为360 MHz，内存128 MB。

506

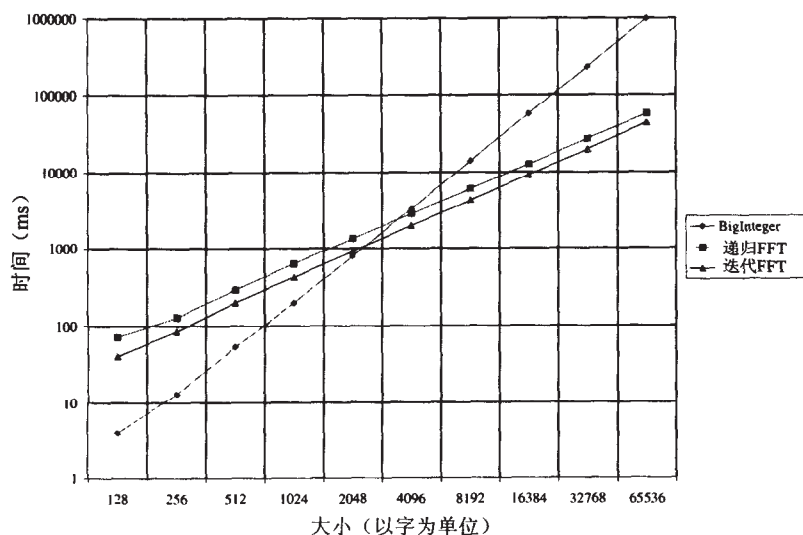


图10-3 整数乘法的运行时间。图显示了整数相乘的运行时间，其中每个整数由若干个字组成，每个字15位（对于FFT是这样），或标准BigInteger实现中同样位数的整数。注意，同时在x轴和y轴上进行了对数比例的调整

注意用log-log比例显示结果。这种选择对应于幂测试(1.6.2节)。与标准线性尺度相比,这样做使得可以更清楚地看到不同实现的相对开销。由于 $\log n^c = c \log n$, log-log比例下的直线的斜率恰好是标准线性尺度下的多项式的幂。同样log-log比例下的直线的高度对应于比例常数。图10-3中的图表展示的时间在y轴上以10为底,在x轴上以2为底。利用 $\log_2 10$ 约为3.322的事实,可见标准乘法算法的运行时间的确为 $\Theta(n^2)$,而FFT算法的运行时间近似为线性。同样,注意到迭代FFT算法实现中的常数因子大约是递归FFT算法实现中常数因子的70%。另外注意到基于FFT的方法和标准乘法算法之间存在显著的折衷。在比例的小端,基于FFT的方法要比标准算法慢10倍,而在比例的高端,基于FFT的方法要比标准算法快10倍!

507

10.6 习题

基础题

- R-10.1 证明定理10.1。
 R-10.2 构造一个类似于表10-1的表,显示方法EuclidGCD(14300, 5915)的执行过程。
 R-10.3 编写算法EuclidGCD的一个非递归版本。
 R-10.4 构造一个类似于表10-1的表,显示方法EuclidBinaryGCD(14300, 5915)的执行过程。
 R-10.5 证明 Z_p 中存在加法逆元,即证明对于每个 $x \in Z_p$,存在 $y \in Z_p$,满足 $x + y \bmod p = 0$ 。
 R-10.6 构造 Z_{11} 中元素的乘法表,表中第 i 行、第 j 列($0 \leq i, j \leq 10$)的元素为 $i \cdot j \bmod 11$ 。
 R-10.7 证明推论10.1。
 R-10.8 给出定理10.5和推论10.1的另一种证明,不能利用定理10.3。
 R-10.9 构造一个类似于表10-4的表,显示方法FastExponentiation(5, 12, 13)的执行过程。
 R-10.10 编写算法ExtendedEuclidGCD的一个非递归版本。
 R-10.11 用两行扩展表10-5,这两行给出了算法每一步中 ia 和 jb 的值,并验证 $ia + jb = 1$ 。
 R-10.12 构造一个类似于表10-5的表,显示方法ExtendedEuclidGCD(412, 113)的执行过程。
 R-10.13 计算数113、114和127在 Z_{99} 中的乘法逆元。
 R-10.14 描述逆FFT算法,它用 $O(n \log n)$ 时间计算逆DFT,即证明如何把 \mathbf{a} 和 \mathbf{y} 的角色互换,并改变赋值语句,使得对于每个输出下标,有

$$a_i = \frac{1}{n} \sum_{j=1}^{n-1} y_j \omega^{-ij}$$

- R-10.15 证明本原单位根的归约性质的更一般形式,即证明对于任意整数 $c > 0$,如果 ω 是第 (cn) 个本原单位根,那么 ω^c 是第 n 个本原单位根。
 R-10.16 对于 $n = 4$ 和 $n = 8$,把第 n 个复数单位根写成 $a + bi$ 的形式。
 R-10.17 对于 $n = 16$,位-逆排列reverse是什么?
 R-10.18 利用FFT和逆FFT计算 $\mathbf{a} = [1, 2, 3, 4]$ 和 $\mathbf{b} = [4, 3, 2, 1]$ 的卷积。像图10-1那样,显示每一项的输出。
 R-10.19 利用卷积定理计算多项式 $p(x) = 3x^2 + 4x + 2$ 和 $q(x) = 2x^3 + 3x^2 + 5x + 3$ 的乘积。
 R-10.20 利用算术模 $17 = 2^4 + 1$,计算向量 $[5, 4, 3, 2]$ 的离散傅里叶变换。利用5是 Z_{17}^* 的一个生成元的事实。
 R-10.21 构造一个表,显示RSA密码系统的一个例子,并且参数 $p = 17$, $q = 19$ 和 $e = 5$ 。表应该有两行,一行用于明文 M ,另一行用于密文 C 。列应该对应 M 的区间 $[10, 20]$ 中的整数值。

508

创新题

- C-10.1 证明二进制Euclid算法(算法10-2)的正确性,并分析它的运行时间。
 C-10.2 设 p 是一个奇素数。
 a. 证明 Z_p 恰好有 $(p-1)/2$ 个二次剩余。
 b. 证明

$$\left(\frac{a}{b}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}$$

- c. 给出查找 Z_p 的一个二次剩余的随机化算法, 要求它的期望时间为 $O(1)$ 。
- d. 讨论散列表中用于解决冲突的二次剩余技术和二次探测技术之间的关系 (见2.5.5节和习题C-2.35)。
- C-10.3 设 p 是一个素数。给出计算 Z_p 中元素的乘法逆元的另一种有效算法, 该算法不能基于扩展欧几里得算法。并分析你所设计算法的运行时间。
- C-10.4 利用对至多为 $2\lceil \log_2 n \rceil$ 位的操作数的算术操作, 证明如何修改算法ExtendedEuclidGCD, 计算 Z_p 中元素的乘法逆元。
- C-10.5 证明用于计算Jacobi符号的方法Jacobi(a, b) (算法10-7) 的正确性。另外, 证明这个方法执行 $O(\log \max(a, b))$ 次算术操作。
- C-10.6 证明如果Rabin-Miller算法的合性证据函数witness(x, n)返回true, 那么数 n 是合数。
- C-10.7 描述一个运行时间为 $O(n^{\log 2^3})$ 的分治算法, 计算阶为 n 且有整数系数的两个多项式相乘。分治算法不能基于FFT算法。假设任何两个整数进行基本算术运算所需时间为常数。
- C-10.8 证明对于任何整数 $b > 0$, 当乘法取模 $(2^{2b} + 1)$ 时, $\omega = 2^{4b/m}$ 是第 m 个本原单位根。
- C-10.9 给定阶为 n 的多项式 $p(x)$ 和 $q(x)$, 描述一个计算 $p'(x) \cdot q'(x)$ 的 $O(n \log n)$ 时间的方法, 其中 $p'(x)$ 和 $q'(x)$ 分别为 $p(x)$ 和 $q(x)$ 的导数。
- C-10.10 描述一个FFT版本的算法, 使其对于 n 为3的幂, 把输入向量分成三个子向量, 递归对每一部分求解, 然后合并这些子问题的解。导出这个算法运行时间的递归方程。并利用主定理理解这个递归方程。
- C-10.11 给定一个实数集合 $X = \{x_0, x_1, \dots, x_{n-1}\}$ 。注意: 根据多项式插值定理, 存在唯一一阶为 $(n-1)$ 的多项式 $p(x)$, 满足 $p(x_i) = 0$, 其中 $i = 0, 1, \dots, n-1$ 。设计一个分治算法, 用 $O(n \log^2 n)$ 时间构造这个 $p(x)$ 的系数表示。

509

程序设计

- P-10.1 编写一个类, 包含模指数运算方法和计算模逆元的方法。
- P-10.2 通过Rabin-Miller和Solovay-Strassen实现随机素性测试算法。对于随机产生的32位整数, 测试这些算法的结果, 其中使用的信心参数分别为7、10和20。
- P-10.3 用Java类实现一个用于整数消息的简化的RSA密码系统, 其中Java类提供了加密、解密、签名和验证的方法。
- P-10.4 用Java类实现一个用于整数消息的简化的El Gamal密码系统, 其中Java类提供了加密、解密、签名和验证的方法。

10.7 本章注记

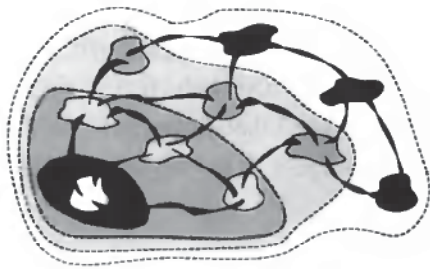
Koblitz[123]和Kranakis[125]的著作对数论知识做了介绍。关于数值算法的经典课本当属Knuth丛书中的第二卷The Art of Computer Programming[121]。Bressoud以及Wagon[40]以及Bach和Shallit[20]的著作中也介绍了数论问题中的算法。Solovay-Strassen随机素性测试算法出现在文献[190, 191]中。Rabin-Miller算法出现在[171]中。

Schneier的著作[180]详细描述了密码协议和算法。Stallings[192]的著作中包含了密码学在网络安全中的应用。RSA密码系统以其三位发明者Rivest、Shamir和Adleman[173]的首字母命名。El Gamal密码系统以其发明者[75]的名字命名。Merkle[153, 154]介绍了散列树结构。单向累加器函数在[27, 179]中介绍。

快速傅里叶变换(FFT)出现在Cooley和Tukey的论文[54]中。Aho、Hopcroft和Ullman[7], Baase[19], Cormen、Leiserson和Rivest[55], Sedgewick[182, 183]以及Yap[213]的著作中也讨论过

它,所有这些著作对上面给出的讨论都产生了影响。尤其是,上面给出的快速整数相乘算法是在Yap和Li提出的QuickMul算法之后形成的。对于FFT的其他应用,有兴趣的读者可进一步参考Brigham[41], Elliott和Rao[66]的著作以及Emiris和Pan[67]的著作中的部分章节。10.1节和10.2节的内容部分基于Achter和Tamassia[1]未发表的手稿中的一节。

第11章



网络算法

网络算法是在计算机网络上执行的算法，这些算法中包括那些指定称为路由器（router）的特殊计算机如何通过网络发送数据包的算法。可把计算机网络模型化一个图，图中顶点表示处理器或路由器，边表示通信信道（如图11-1所示）。运行在这种网络上的算法不是常规算法，因为这些算法的输入通常遍及整个网络，并且这些算法的执行分布在网络中的处理器之间。

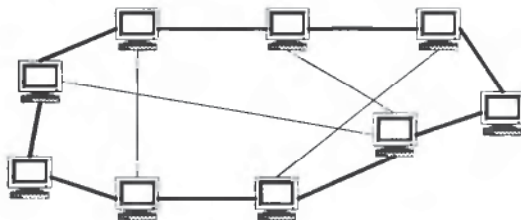


图11-1 计算网络，它的顶点是处理器，边是通信链路。粗边定义了一个简单的环，称为环（ring）网

我们从考虑几个基本的分布式算法开始学习网络算法。这些算法提出了网络算法中遇到的一些基本问题，包括领导人选举问题和构造生成树问题。我们同时以同步和异步方式考察每个算法，在同步方式中，处理器可以工作在多“轮”的“锁定步骤”中；在异步方式中，处理器可以以相对不同的速度运行。在某些情况下，处理异步情况引入了新的复杂问题，但在各种网络（如因特网）中，异步情况是一个必须有效处理的现实情况。

路由（routing）算法是网络算法中的一大类。它指定如何在网络中的各台计算机之间移动信息包。良好的路由算法应该把数据包快速、可靠地路由到它的目的地，同时还要兼顾网络中其他数据包的“公平”。设计一个良好的路由算法面临的挑战是这些目标有时是相互矛盾的，因为公平性有时与快速包传输相冲突。另一个挑战是实现路由模式所需的处理过程也应该快速和高效。因此，本章第二部分集中讨论路由算法，包括下列通信模式的方法：

- 广播路由（broadcast routing），把一个数据包发送给每台计算机
- 单播路由（unicast routing），把一个数据包发送给特定的计算机
- 多播路由（multicast routing），把一个数据包发送给一组计算机

通过分析路由算法中涉及的代价，包括设置算法的代价，以及利用那个算法路由消息的代价，来研究这些算法。但是，在讨论这些网络算法和其他网络算法如何工作之前，首先应该解释算法

运行所依托的计算模型。

512

11.1 复杂性测度和模型

在能够充分研究网络算法之前，需要更好地理解网络（如因特网）如何工作。确切地讲，要探索处理器间的通信如何进行、模型化和进行分析。

11.1.1 网络协议栈

讨论网络（如因特网）功能的一种方式是根据分层的网络模型，该模型具有如下概念上的层：

- **物理层 (physical layer)**：在这一层上原始位沿着某些介质（如电线或光缆）传输。这一层的设计决策主要涉及表示1和0的工程技术，最大化传输带宽以及最小化传输噪声。
- **数据链路层 (data-link layer)**：这一层处理把数据分解成子单元的方法，称子单元为帧 (frame)，并通过物理层把数据从一台计算机传输到另一台计算机。典型例子包括点对点协议，这个协议描述了如何在两台物理连接的计算机之间发送数据。
- **网络层 (network layer)**：这一层处理把数据分解成包 (packet)（不同于帧）的方法，并通过整个网络把这些包从一台计算机路由到另一台计算机。这一层涉及的算法上的问题有：在网络中路由包的方法以及分解包和合并包的方法。这一层使用的主要因特网协议是网际协议 (Internet protocol) 或IP。这个协议利用“尽力传输”方法，这意味着性能得不到保证，某些包可能会丢失。
- **传输层 (transport layer)**：这一层从各种应用中接收数据，如果需要，把它分解成更小的单元，并把它们传递到网络层，这一层能够保障特定的可靠性和拥塞控制性能。这一层里有两个主要网际协议，一个是传输控制协议 (transmission control protocol, TCP)，它提供因特网上两台机器之间无差错、端到端、面向连接的传输。另一个协议是用户数据报协议 (user datagram protocol, UDP)，它提供一种无连接、尽力传输的方法。
- **应用层 (application layer)**：应用运行在这一层里，它使用较低级的协议。因特网应用的例子包括电子邮件 (SMTP)、文件传输 (FTP和SCP)、虚拟终端 (TELNET和SSH)、万维网 (HTTP) 和域名服务 (DNS)，DNS将主机名（如www.cs.brown.edu）映射到IP地址（如128.148.32.110）。

513

11.1.2 消息传递模型

在这一章里，几乎没有讨论网络协议栈的细节。而是讨论工作在计算模型上的网络算法。这个计算模型把各层提供的功能抽象在上述网络协议栈 (network protocol stack) 中。对网络协议栈各层的理解使我们可以把分布式计算模型中的概念绑定到真实网络（如因特网）中的特定功能和协议上。

抽象网络功能的方式有多种，可以以一种允许我们描述网络算法的方式来进行。在这一章里，利用普遍使用的消息传递模型描述这样的分布式算法 (distributed algorithm)，处理器之间的通信是松散耦合的，目的是解决计算上的问题。14.2节介绍了并行算法 (parallel algorithm) 的另一种模型。在一个并行算法中，紧密耦合的多处理器通过共享存储空间进行合作和协调。

本章唯一关注的是分布式消息传递模型，在这个模型中把网络模型化为一个图，图中顶点表示处理器，边表示处理器之间的固定连接，它们在计算过程中不能改变。每条边 e 支持关联 e 端点的两个处理器之间的消息传递。如果 e 是有向的，则消息只能在一个方向发送。例如，在因特网

的网络层，顶点对应称为IP路由器（IP router）的特殊计算机，每台计算机由唯一数值码标识，称之为IP地址（IP address），路由器交换的消息是称为IP包（IP packet）的基本数据传输单元。另外，因特网上的顶点也可对应服务器，边对应维持各对服务器之间的永久TCP/IP连接。

顾名思义，在消息传递模型中，通过在网络上交换消息实现信息通信。给网络中的每个处理器分配唯一的数值标识符（identifier）（例如，可以是因特网上路由器的IP地址或主机的IP地址）。此外，假设每个处理器知道它在网络上的近邻，并且它只与其近邻直接通信。但在许多情况下，这个全局知识是不可用的或不必要的。

网络算法中考虑的另一个重要考虑是当算法运行时，能否处理网络中的潜在变化。在算法执行过程中，处理器可能发生故障（或“损毁”），网络连接可能损坏，某些处理器可能保持活动状态，但进行错误或恶意计算。理想情况下，希望分布式算法是动态的，即能够响应网络中的变化情况，并且能容错，即能够正确优雅地从故障中恢复。然而，使分布式算法完全健壮的技术相当复杂；因此，它们超出了本书的范围。于是，在这一章里，假设网络拓扑结构不变，处理器处于活动状态，并且在网络执行一个分布式算法时没有故障发生，即假设在算法执行过程中，网络是静态的。

在消息传递模型中，进行分步计算考虑最重要的因素之一是假设网络中的处理器是同步的。关于同步可以做出几种选择。但以下模型是分布式算法设计中使用最多的两个模型：

- 同步模型（synchronous model）：在这个模型中，每个处理器都有一个内部时钟，记录程序执行时间，并且所有处理器的时钟是同步的。此外，假设处理器的执行速度是一致的，每个处理器执行相同操作（如加法操作或比较操作）的时间也一样长。最后，假设通过网络中的任意连接发送一条消息的时间相同。
- 异步模型（asynchronous model）：在这个模型中，假设处理器没有内部时钟。此外，并不假设处理器的速度必须相似。因此，异步算法中的每一步由条件或事件（event）确定，而不是由时钟“嘀嗒次数”确定。尽管如此，还可做出一些合理的计时假设，使算法有效执行任务。首先，假设用边表示的每条通信信道是一个先进先出（FIFO）队列，可以缓冲任意数量的消息。也就是说，在一条边上发送的消息可存储在那条边上的一个缓冲区中，使消息到达次序与发送次序保持一致。其二，假设当处理器速度可以变化时，这种速度的变化不是任意的。也就是说，有一个基本的公平性（fairness）假设，保证如果处理器 p 中有一个事件使 p 能够执行一个任务，那么 p 最终将会执行这个任务。

除了分布式计算模型的这两个截然不同的版本之外，还可以考虑一些中间版本，这些版本的模型假设处理器中有时钟，但这些时钟可能不完全同步。因此，在这样一个中间模型中，不能假设处理器计算按照“锁定步骤”进行，而是假设处理器基于“超时”事件做出选择，此后很长一段时间处理器未做出响应。尽管这样的中间模型与现实网络（如因特网）的实际计时性质更相近，但我们仍然会把注意力局限到上述的同步模型和异步模型上。为同步模型所设计的算法通常相当简单，可以改变成在异步模型上执行的算法。此外，可以肯定的是，为异步模型所设计的算法仍然可在一个具有有限同步的中间模型上执行。

11.1.3 网络算法的复杂性测度

在传统算法设计中，如在本书其他章中研究的那样，确定一个算法效率所用的复杂性测度是所用的运行时间和内存空间。但是，这些复杂性测度并不能直接变成网络算法中的测度，因为它们蕴涵着一个假设，即计算是在一台计算机上进行的，而不是在计算机网络中进行的。在网络算法中，输入在网络中的计算机上传播，并且计算在网络中的多台计算机上进行。因此，必须仔细

考虑表征网络算法的性能参数。

与传统算法相比,分布式算法中的复杂性测度具有自然的相似之处,但网络中还有其特有的复杂性测度。在这一章里,集中考虑以下复杂性测度:

- **计算轮 (computational round):** 通过一系列全局轮执行若干网络算法,最后收敛到一个解。收敛所需的轮数可用作时间的大致近似。在同步算法中,这些轮可由时钟嘀嗒数确定,而在异步算法中,这些轮常常由穿过网络的事件传播“波”确定。
- **空间 (space):** 一个计算所需的空間可用于网络算法的测度,但必须确定是作为算法中所有计算机使用总空间的全局界限,还是作为涉及某台计算机所需的局部空间量。
- **局部运行时间 (local running time):** 分析一个计算所需的全局运行时间是困难的,尤其是对于异步算法则更是如此。我们仍可以分析网络算法中参与的某台计算机所需的局部计算时间。如果算法中的所有计算机在本质上执行相同类型的功能,那么一个局部运行时间的界限就足够了。但如果在算法中有几类不同类型的计算机参与,那么就应该表征每类计算机所需的局部运行时间。
- **消息复杂性 (message complexity):** 这个参数对计算中所有计算机对之间发送的消息总数(消息大小至多为处理器数的对数函数)进行度量。例如,如果通过从一台计算机到另一台计算机的 p 条边路由一条消息 M ,则称这个通信的消息复杂度是 $p|M|$,其中 $|M|$ 表示 M 的长度(以字为单位)。

最后,如果在试图优化的一个网络算法(如在线拍卖协议的利润或在线高速缓存模型的检索代价)时存在其他一些明显的复杂性测度,那么这些测度也应该包括在这个复杂性的界限中。

516

11.2 基本分布式算法

网络算法的复杂性测度常常使人想起正在试图求解的问题“规模”的某些直观概念的作用。例如,可按如下参数表示问题的规模:

- 表示输入的字数。
- 部署的处理器数。
- 处理器之间的通信连接数。

因此,为了正确地分析网络算法的复杂性测度,必须对那个算法求解问题规模的含义进行形式化。

为了使这些概念更具体,在这一节里研究几种基本的分布式算法。利用上述复杂性测度分析它们的性能。选择进行研究的特定问题既说明了基本分布式算法的技术,也将作为分析网络算法的方法。

11.2.1 环网上的领导人选举

我们讨论的第一个问题是环网中的领导人选举 (leader election) 问题。在这个问题中,给定 n 个处理器的一个网络,处理器连接成环,即网络的图是一个具有 n 个顶点的简单回路。目标是选出这 n 个处理器中的一个作为领导人,并使所有其他处理器赞同这个选举。我们描述了环是有向图的这种情况的一个算法。习题C-11.1考虑环是无向图的情况。

首先描述该问题的一个同步解。同步解的主要思想是选择具有最小标识符的处理器作为领导人。面临的挑战是在环网上没有明显的开始位置,因此,可在任何地方开始。在算法的一开始,每个处理器把它的标识符发给环网中的近邻处理器。在后续的轮中,每个处理器执行以下

计算:

- (1) 从它在环网上的前驱接收一个标识符 i 。
- (2) 把 i 与自己的标识符进行比较。
- (3) 把这两个值中的最小值发给它在环网中的后继。

如果一个处理器从前驱接收到自己的标识符,那么这个处理器知道它必定有最小标识符,因此,它是领导人。然后这个处理器沿着环网发送消息,通知所有其他处理器它是领导人。

517

算法11-1中给出了上述方法的描述。算法的执行示例如图11-2所示。

算法11-1 有向环网上每个处理器运行的同步算法,用于确定环网中处理器的“领导人”

```

算法 RingLeader( $id$ ):
  输入: 运行此算法的处理器唯一标识符 $id$ 
  输出: 环网中处理器的最小标识符
   $M \leftarrow [\text{Candidate is } id]$ 
  把消息 $M$ 发送给环网中的后继处理器
   $done \leftarrow \text{false}$ 
  repeat
    从环网中的前驱处理器得到消息 $M$ 
    if  $M = [\text{Candidate is } i]$  then
      if  $i = id$  then
         $M \leftarrow [\text{Leader is } id]$ 
         $done \leftarrow \text{true}$ 
      else
         $m \leftarrow \min\{i, id\}$ 
         $M \leftarrow [\text{Candidate is } m]$ 
      else
        { $M$ 是一个“Leader is”消息}
         $done \leftarrow \text{true}$ 
    把消息 $M$ 发送给环网中的下一个处理器
  until  $done$ 
  return  $M$     { $M$ 是一个“Leader is”消息}
  
```

在分析算法RingLeader之前,首先确信算法正确工作。设 n 是处理器数量。在第一轮中每个处理器发送它的标识符。然后,在下面的 $n-1$ 轮中,每个处理器接收一个“candidate is”消息,计算这个消息中标识符 i 的最小值 m 和它自己的 id ,并把标识符 m 与“candidate is”消息一起传输到下一个处理器中。设 ℓ 是环网中某个处理器的最小标识符,处理器 ℓ 发送的第一条消息会遍历整个环网,最终不变地返回到处理器 ℓ 中。在那个时刻,处理器 ℓ 将认识到它是领导人,并用消息“leader is”通知所有其他处理器。在接下来的 n 轮中,这条消息将遍历整个环网。应该注意到上述分析中使用的处理器数量 n 并不为处理器所知。同样,观察可见算法中不存在死锁(deadlock),即不存在两个处理器相互等待消息的情况。

现在分析算法RingLeader的性能。首先,按照轮数,来自领导人的第一条“candidate is”消息需要 n 轮遍历环。此外,领导人发起的消息“leader is”也需要 n 轮才能到达所有其他处理器。

518

因此,有 $2n$ 轮。

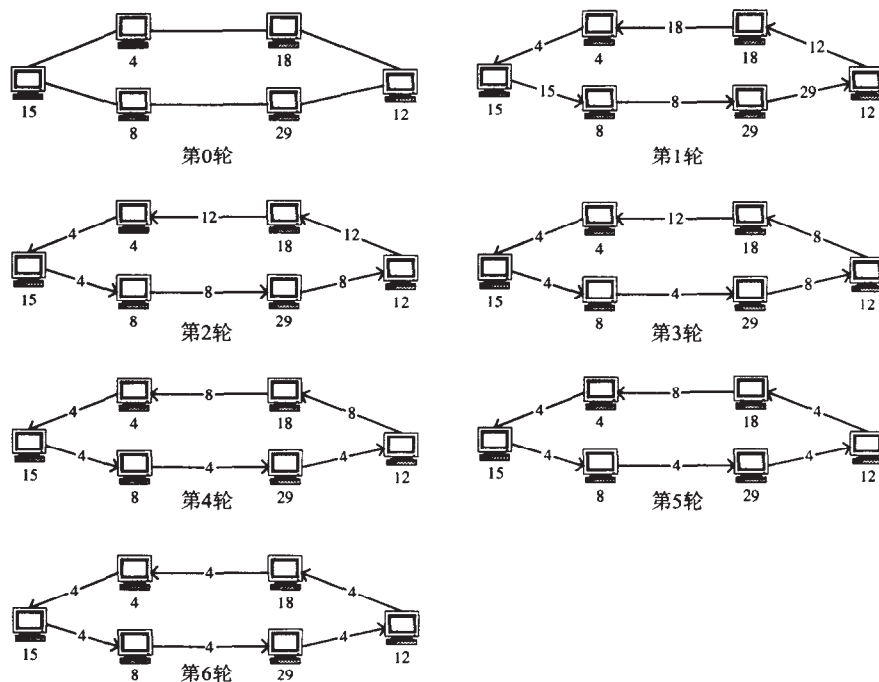


图11-2 环网上同步领导人选举算法的说明。初始配置标记有“第0轮”。对于后续每一轮，每条边上标记有沿着这条边发送的“candidate is”消息。没有显示最后一轮，在这一轮中处理器4通知其他处理器，它就是具有“leader is”消息的领导人

现在分析算法的消息复杂度。把算法分成两个阶段。

(1) 第一个阶段由前 n 轮组成，在这些轮中处理器发送“candidate is”消息，并继续传播它们。在这一阶段中，每一轮中每个处理器发送和接收一条消息，因此，第一阶段发送 $O(n^2)$ 条消息。

(2) 第二阶段由后 n 轮组成，“leader is”消息沿着环行进。在这一阶段中，处理器继续发送“candidate is”消息，直到“leader is”消息到达它。此时，它转发“leader is”消息并停止。因此，在这一阶段中，领导人发送一条消息，领导人的后继发送两条消息，后继的后继将发送三条消息，依此类推。因此，第二阶段中发送消息的总数是 $\sum_{i=1}^n i$ ，它就是 $O(n^2)$ 。

519

总结如下。

定理11.1 给定 n 个结点的分布有向环网 N ，其中结点的标识符互不相同，但没有明显的领导人。RingLeader算法在 N 中找到一个领导人，总共利用 $O(n^2)$ 条消息。此外，算法的总消息复杂度为 $O(n^2)$ 。

证明 我们已经看到发送了 $O(n^2)$ 条消息。因为每条消息利用 $O(1)$ 个字，因而消息总复杂度为 $O(n^2)$ 。 ■

可能改进环网上领导人选举的消息复杂度。修改算法RingLeader，使它需要更少的消息。在习题C-11.2和C-11.3中探讨了两种可能的修改方法。

异步领导人选举

在同步模型假设之下，我们描述和分析了算法11-1（RingLeader）的性能。此外，我们可以构造每一轮，使它由消息接收步骤、处理步骤和消息发送步骤组成。这就是典型的同步分布式算法的结构。

在异步算法中，我们不再假设处理器以“锁定步骤”方式移动。而是通过事件决定处理过程，

而不过时钟决定。上述环网中领导人选举算法仍然能在异步模式下工作。算法的正确性实际上并不依赖于处理器的同步作用。它只依赖于每个处理器按照发送的次序从它的前驱接收消息。这个条件在异步模型中仍然成立。

对环网上领导人选举的研究概括为如下定理。

定理11.2 给定 n 个处理器的有向环网，算法RingLeader执行领导人选举的消息复杂度为 $O(n^2)$ 。对于每个处理器，局部计算时间是 $O(n)$ ，局部所用空间是 $O(1)$ 。算法RingLeader工作在同步模型和异步模型下。在同步模型中，它的总运行时间为 $O(n^2)$ 。

领导人选举问题起初看起来似乎是一个人为的问题，但它实际上有许多应用。任何时间如果需要计算网络中结点子集的一个全局功能，实质上就必须进行领导人选举。例如，计算存储在环网连接的计算机中的值之和，或者求这些值的最小值或最大值，都可利用一个类似上述给出的领导人选举算法。

当然，处理器并不总是按照环网连接。因此，在下一节里，我们探索处理器按照（自由）树结构连接的情况。

520

11.2.2 树网上的领导人选举

如果网络是一棵（自由）树，那么选举领导人要比在环网上简单得多。因为树结构自然有开始计算的地方：即外部结点。

1. 树网上的异步领导人选举

树网上的异步领导人选举算法如算法11-2所示。再次选择具有最小标识符的处理器作为领导人。算法假设处理器能够对于所依附的边进行常数时间的消息检查（message check），如果消息已从那条边到达的话。

算法11-2 在处理器的树网上计算领导人的算法。在算法的同步版本中，所有处理器在同一时刻开始执行，直到停止

算法 TreeLeader(id):

输入：运行此算法的处理器唯一标识符 id

输出：树网中处理器的最小标识符

{累积阶段}

设 d 是处理器 id 的近邻数 $\{d \geq 1\}$

$m \leftarrow 0$ {接收消息的计数器}

$\ell \leftarrow id$ {暂定的领导人}

repeat

{开始新一轮}

for 每个近邻 j **do**

检查来自处理器 j 的消息是否已到达

if 来自 j 的消息 $M = [\text{Candidate is } i]$ 已经到达 **then**

$\ell \leftarrow \min\{i, \ell\}$

$m \leftarrow m + 1$

until $m \geq d - 1$

if $m = d$ **then**

$M \leftarrow [\text{Leader is } \ell]$

for 每个近邻 $j \neq k$ **do**

把消息 M 发送给处理器 j

```

return  $M$     { $M$ 是“leader is”消息}
else
     $M \leftarrow$  [Candidate is  $\ell$ ]
    把消息 $M$ 发送给还未发送消息的近邻 $k$ 
    {广播阶段}
repeat
    {开始新一轮}
    检查来自处理器 $k$ 的消息是否已到达
    if 来自 $k$ 的消息 $M$ 已经到达 then
         $m \leftarrow m + 1$ 
        if  $M =$  [Candidate is  $i$ ] then
             $\ell \leftarrow \min\{i, \ell\}$ 
             $M \leftarrow$  [Leader is  $\ell$ ]
            for 每个近邻 $j$  do
                把消息 $M$ 发送给处理器 $j$ 
        else
            { $M$ 是“leader is”消息}
            for 每个近邻 $j \neq k$  do
                把消息 $M$ 发送给处理器 $j$ 
    until  $m = d$ 
return  $M$     { $M$ 是“leader is”消息}

```

算法TreeLeader的主要思想是利用两个阶段。在累积阶段（accumulation phase）中，标识符从树的外部结点流进。每个结点记录从其近邻收到的最小标识符 ℓ ，并在它收到除一个近邻标识符之外的所有标识符后，把标识符 ℓ 发送给那个近邻。在某个点上，结点接收到所有近邻的标识符。称这个结点为累积结点（accumulation node），它确定领导人。图11-3说明了这个累积阶段。一旦某个结点上有标识过的领导人，就开始广播阶段。在广播阶段中，累积结点把领导人的标识符向外部结点广播。注意，在两个近邻结点都成为累积结点时，可能出现不分胜负的情况。在这种情况下，它们向各自树的“一半”广播。树中的任何结点（即使是一个外部结点）都可能是累积结点。这取决于处理器的相对速度。

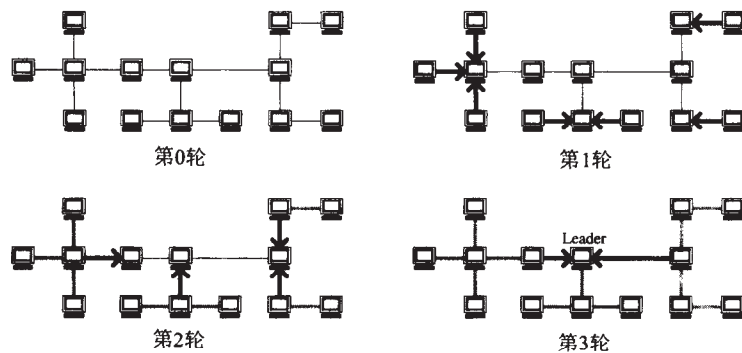


图11-3 树网上查找领导人的算法说明。只显示了累积阶段。尚未携带消息的边用细线表示。携带那一轮中流动消息的边用粗箭头画出。灰线表示在前一轮中已经传输过消息的边

2. 树网上的同步领导人选举

在算法的同步版本中，所有处理器在同一时刻开始一轮。因此累积阶段的消息流进累积结点，

就像是一块石头掉进一池水中引起的逆向波纹。同样，在广播阶段，消息从累积结点向外传播，就像是池中波纹的扩散过程，即在累积阶段消息行进到树的“中心”，在广播阶段消息向外扩散出来。

图的直径（diameter）是图中任意两个结点之间的最长路径长度。对于树，直径是在两个外部结点之间的路径长度。有趣的是，在算法的同步版本中，轮数恰好等于树的直径。

3. 树网领导人选举算法的性能

分析树网异步领导人选举算法的消息复杂度相当直观。在累积阶段，每个处理器发送一条“candidate is”消息。在广播阶段，每个处理器至多发送一条“leader is”消息。每条消息大小为 $O(1)$ 。因此，消息复杂度是 $O(n)$ 。

现在研究同步算法的局部运行时间。假设忽略等待下一轮开始所花费的时间。处理器 i 上的算法花费的时间为 $O(d_i D)$ ，其中 d_i 是处理器 i 的近邻数， D 是树的直径。同样，处理器 i 利用 d_i 的空间记录已经发送消息的近邻。

对树网上领导人选举的研究概括为如下定理。

定理11.3 给定有 n 个结点、直径为 D 的一棵（自由）树，算法TreeLeader执行领导人选举的消息复杂度为 $O(n)$ 。算法TreeLeader工作在同步模型和异步模型下。在同步模型中，对于每个处理器，局部计算时间是 $O(d_i D)$ ，局部所用空间是 $O(d_i)$ ，其中 d_i 是处理器 i 的近邻数。

11.2.3 广度优先查找

假定有一个处理器连接的通用网络，我们希望标识这个网络中的某个特定顶点 s 。在6.3.3节中，讨论从某个源点 s 开始，构造图 G 广度优先查找的一个集中式算法。在这一节里，描述求解这个问题的一个分布式算法。

523

1. 同步BFS

首先描述一个简单的同步广度优先查找（BFS）算法。算法的主要思想是按照“波动”过程进行，从源点开始向外传播，自顶向下一层一层地构造一棵BFS树。处理器的同步使我们大大受益。在这种情况下，使传播过程完全协调。

在当前BFS树中，首先把 s 作为“外部结点”开始算法过程，此时只有结点 s 。然后，在每一轮中，每个外部结点 v 把一条消息发送到早先还未联系 v 的它的所有近邻中，通知它们 v 想使它们成为BFS树中 v 的子结点。如果这些结点还未选择其他结点作为它们的父结点，则它们通过选择 v 作为父结点来响应。

图11-4说明了这个算法。算法11-3中给出了它的伪代码。

算法11-3 在一个连通的处理器网络中计算一棵广度优先查找树的同步算法

算法 SynchronousBFS(v, s):

输入: 执行此算法的结点（处理器）的标识符 v ，BFS遍历的起始结点的标识符 s

输出: 对于每个结点 v ，它在以 s 为根的BFS树中的父结点

repeat

{开始新一轮}

if $v = s$ **or** v 已经接收到来自其中一个近邻的消息 **then**

 设置parent(v)为请求 v 作为其子结点的那个结点（或null，如果 $v = s$ ）

for每个与 v 近邻且未联系 v 的结点 w **do**

 向 w 发送消息，请求 w 成为 v 的一个子结点

until $v = s$ **or** v 已收到消息

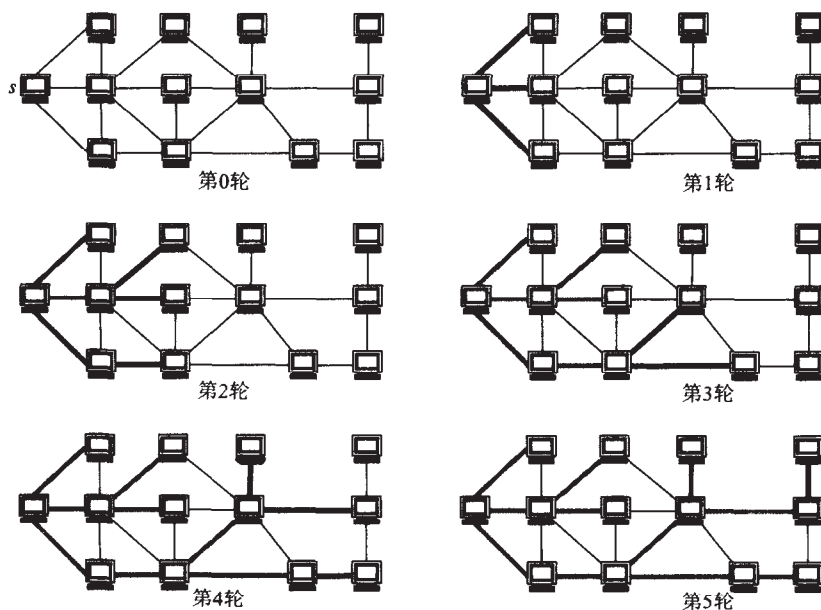


图11-4 同步BFS算法的说明。粗边就是BFS树中的那些边，黑粗边是在当前轮中被选择的边

对算法SynchronousBFS的分析是简单的。在每一轮中，向外传播到BFS树的另一层。因此，这个算法的运行时间与BFS树的深度成正比。此外，在整个计算的每个方向的每一条边上至多发送一条消息。因此，在具有 n 个结点、 m 条边的网络中发送的消息数为 $O(n+m)$ 。因此，这个算法相当有效。但可以肯定这个算法利用了处理器的同步机制。

2. 异步BFS

可以保存需协调事件的其他信息，使上述BFS算法成为异步算法。同样，要求每个处理器知道网络中的处理器总数。

我们不是依赖于同步时钟来确定计算中的每一轮，而是从一个源点 s 开始向外发送“脉冲”消息，触发其他处理器执行计算的下一轮。利用这项脉冲技术，仍然可以使计算一层一层地向外传播。脉冲过程自身就像是呼吸——有一个向下脉冲阶段（pulse-down phase），信号沿着BFS树的根 s 向下传递，还有一个向上脉冲阶段（pulse-up phase），从外部结点组合信号返回到BFS树的根 s 。

如果外部结点上的处理器已经从它们各自的父结点接收到一个新的向下脉冲信号，它们只能从一轮到下一轮。同样，根 s 在接收到它的子结点的所有向上脉冲信号后，才会发出新一轮的向下脉冲信号。这样，就能保证处理器处于近似同步状态（粒度为轮级的同步）。算法11-4中描述了同步BFS算法的细节。

算法11-4 在一个连通的处理器网络中计算一棵广度优先查找树的异步算法

算法 AsynchronousBFS(v, s, n):

输入：运行这个算法的处理器标识符 v ，BFS遍历的起始结点的标识符 s ，以及网络中的结点数 n

输出：对于每个结点 v ，它在根为 s 的BFS树中的父结点

$C \leftarrow \emptyset$ {经验证的 v 在BFS树中的子结点}

设 A 为 v 的近邻集合 { v 的BFS树的候选子结点}

repeat

{开始新一轮}


```

if parent( $v$ )已定义 or  $v = s$  then
  if parent( $v$ )已定义 then
    等待来自parent( $v$ )的pulse-down消息
  if  $C$ 非空 then
    { $v$ 是BFS树中的一个内部结点}
    把pulse-down消息发送到 $C$ 中的所有结点中
    等待来自 $C$ 中所有结点的pulse-up消息
  else
    { $v$ 是BFS树中的一个外部结点}
    for  $A$ 中的每个结点 $u$  do
      向 $u$ 发送make-child消息
    for  $A$ 中的每个结点 $u$  do
      从 $u$ 得到消息 $M$ , 并从 $A$ 中删除 $u$ 
      if  $M$ 是一条accept-child消息 then
        把 $u$ 添加到 $C$ 中
    向parent( $v$ )发送pulse-up消息
  else
    { $v \neq s$ 还没有父结点}
    for  $A$ 中的每个结点 $w$  do
      if  $w$ 已经向 $v$ 发送一条make-child消息 then
        从 $A$ 中删除 $w$  { $w$ 不再是 $v$ 的候选子结点}
      if parent( $v$ )未定义 then
        parent( $v$ )  $\leftarrow w$ 
        向 $w$ 发送一条accept-child消息
      else
        向 $w$ 发送一条reject-child消息
    until ( $v$ 接收到消息done) or ( $v = s$ 且执行 $n-1$ 次向下脉冲)
    把done消息发送给 $C$ 中的所有结点

```

526

3. 性能

这个异步BFS算法肯定比它的同步算法更复杂。然而计算仍然按照一系列的轮操作。在每一轮中, 根结点 s 沿着部分构造的BFS树向下发送“pulse-down”消息。当这些消息到达树的外部结点层时, 当前的外部结点通过向候选子结点发出“make-child”消息, 试图把BFS树再扩展一层。当这些候选结点响应时, 要么接受作为子结点的邀请要么拒绝, 处理器发送“pulse-up”消息, 传播回 s 。然后根 s 一遍又一遍地开始整个过程。

因为BFS树高度至多有 $n-1$ 个结点, 结点 s 重复脉冲行为 $n-1$ 次, 确信网络中的每个结点都包括在BFS树中。因此, 完全利用消息传递完成从一轮到另一轮的同步。当前外部结点在得到一条“pulse-down”消息后, 才开始操作。且当前外部结点在得到其所有候选子结点的返回消息后(要么接受作为子结点的邀请要么拒绝), 才会发布“pulse-up”消息。BFS一层一层地增长, 正像在同步算法中那样。

然而执行这个协调所需的所有消息数量比同步算法要多。网络中的每条边仍然至多只有一条“make-child”消息, 每个方向都有一个响应。因此, 接受和拒绝make-child请求的消息总复杂度为 $O(m)$, 其中 m 是网络中的边数。但在算法的每一轮中, 需要传播所有的“pulse-down”消息和“pulse-up”消息。因为它们只沿着BFS树中的边遍历, 每一轮中至多发送 $O(n)$ 个向上脉冲消息和向下脉冲消息。给定 $n-1$ 轮的源操作, 协调脉冲至多发布 $O(n^2)$ 条消息。于是, 算法的消息总复杂

度为 $O(n^2 + m)$ ，其中 m 是网络中的边数。因 m 为 $O(n^2)$ ，因此这个界限可以进一步简化为 $O(n^2)$ 。

因此，上述讨论可概括为如下定理。

定理11.4 给定具有 n 个顶点、 m 条边的网络，基于同步模型计算 G 的一棵广度优先生成树只用 $O(n^2)$ 条消息。

在习题C-11.4中，我们探索了如何把异步广度优先查找算法的运行时间改进到 $O(nh + m)$ ，其中 h 是BFS树的高度。

527

11.2.4 最小生成树

回忆加权图的一棵最小生成树（minimum spanning tree, MST）是一个生成子图，它是一棵树且有最小权值（7.3节）。在这一节里，描述此问题的一个有效分布式算法。在提供细节之前，首先回顾Barůvka提出的找一棵MST的有效顺序算法（7.3.3节）。算法从 G 中的每个结点所在（零）树边的连通分量开始，然后按照轮进行，把树边添加到连通分量中，从而构造一棵MST树。与所有MST算法一样，它基于如下事实：即对于把顶点分成两个非空子集的任何划分 $\{V_1, V_2\}$ ，如果 e 是连接 V_1 和 V_2 的最小加权边，那么 e 属于一棵最小生成树（见定理7.8）。

为简明起见，假设所有边上权值是唯一的。回忆可知，起初，图中每个结点都在自己的（平凡）连通分量中，树边的初始集 T 为空。在较高级别，Barůvka算法执行如下计算：

```
while 存在 $T$ 定义的多个连通分量 do
  for 每个连通分量 $C$ 并行do
    选择连接 $c$ 与另一连通分量的最小边 $e$ 
    把 $e$ 添加到树边的集合 $T$ 中
```

对于每个连通分量，Barůvka算法并行执行上述计算。因而能够快速找到树边。事实上，注意在每一轮中，保证把每个连通分量至少与另一个连接起来。因此，在每一轮中（由while循环的迭代次数定义）连通分量数会下降2倍。这蕴涵着轮数是顶点数的对数函数。

分布式Barůvka算法

基于同步模型，我们实现分布式Barůvka算法（这个算法的异步模型版本留作习题C-11.7）。在每一轮中，需要进行两个关键的计算——确定所有连通分量，且需要确定每个连通分量的最小出边（回忆我们假设边上权值是唯一的，因而每个分量的最小出边也将是唯一的）（见图11-5）。

假设对于每个结点 v ， v 存储 T 中依附于 v 的边的一个表。因此，每个结点属于一棵树，但是 v 只存储 T 中它的近邻信息。在每一轮中，把11.2.2节中树网上的领导人选举算法作为辅助方法使用两次：

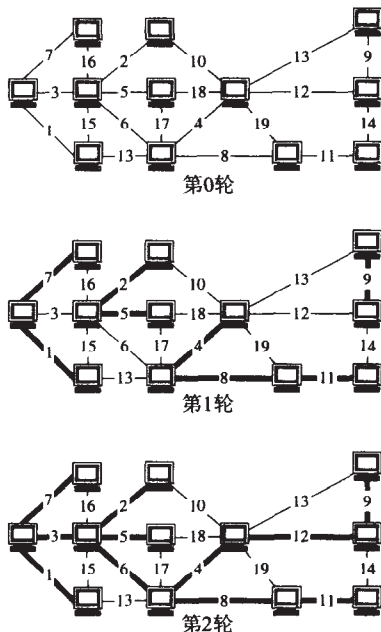
- (1) 确定每个连通分量。
- (2) 对于每个连通分量，找出连接这个连通分量与另一分量的最小权值边。

528

轮操作如下。为了确定连通分量，利用每个结点的标识符进行领导人选举计算。因此，每个分量用其结点的最小标识符标识。接下来，每个结点 v 计算依附 v 的最小权值边 e ，满足 e 的端点在不同的连通分量中。如果不存在这样的边，利用一个无限权值的虚拟边。然后利用关联每个顶点及其权值的边，再次进行领导人选举，产生每个连通分量 C 的最小权值边，这条边把 C 和另一个分量连接起来。注意当检测到一轮中计算的最小权值边上的权值为无穷大时，算法可以终止。

设 n 和 m 分别表示网络中的结点数和边数。为了分析分布式Barůvka算法的消息复杂度，注意每一轮中至多发送 $O(m)$ 条常数大小的消息。因此，由于有 $O(\log n)$ 轮，消息总复杂度为 $O(m \log n)$ 。在习题C-11.6中，考察了一种改进方式，把同步MST的消息复杂度改进到 $O(m + n \log n)$ 。且在习

题C-11.7中, 给读者提出了一个挑战性的问题, 要求找出一个异步MST算法, 使其消息复杂度仍然为 $O(m \log n)$ 。



529

图11-5 Baruvka算法的同步分布式版本的说明。粗边确定MST的边, 黑粗边是给定轮中选择的边

11.3 广播路由和单播路由

在研究了一些基本的分布式算法之后, 现在转到通信网络中的两个公共问题——广播路由 (broadcast routing) 问题和单播路由 (unicast routing) 问题。广播路由是从一个处理器把一条消息发送给网络中的所有其他处理器。当某个处理器中的一条消息要被网络中的所有其他处理器共享时, 需要利用这种通信形式。另一方面, 单播路由包括在网络中建立数据结构, 以便支持点对点通信, 在这里某个处理器中有一条消息, 希望把它中继给另一个处理器。

当讨论本节其余部分的路由算法时, 把处理器称为路由器 (router)。回忆一下假设网络是固定的, 且不随时间改变。因此, 我们只研究静态路由算法。还假设路由器进行消息交换的时间为常数, 这蕴涵着消息复杂度与交换的消息总数成正比。本节始终假设 n 和 m 分别表示网络中的结点数和边数。

我们开始讨论广播路由问题, 然后介绍单播路由和多播路由问题。本节提出的算法是因特网上所用的分组路由的简化版本。

11.3.1 广播路由的洪泛算法

广播路由的洪泛 (flooding) 算法相当简单, 事实上不要求设置, 但它的路由代价高昂。

希望把消息 M 发送到网络中所有其他路由器中的路由器 s 一开始会简单地把消息 M 发给它的近邻。当路由器 $v \neq s$ 接收到来自近邻路由器 u 的一条洪泛消息 M 时, v 简单地重新把 M 广播给它的所有近邻, 除了 u 自身之外。当然, 如果不加修改, 这个算法对于包含回路的网络, 将会引起消息的“无限循环”; 然而, 正如在习题 (C-11.5) 中探讨的那样, 没有回路的网络具有简单的路

由算法。

1. 带有跳数计数器的洪泛

为了避免无限循环问题，必须在洪泛算法中关联主要角色一些记忆或状态 (state)。一种可能性是在每条消息中增加一个跳数计数器 (hop counter)，每当一个路由器处理 M 时，就使 M 的跳数计数器减1。如果跳数计数器到0，那么抛弃它所属于的消息。如果初始化消息 M 的跳数计数器为网络的直径，那么消息就能到达所有路由器，同时避免创建无限个消息。

530

2. 带有序列号的洪泛

另一种可能性是在每个路由器中存储一个散列表 (或另一个有效的字典数据结构)，记录这个路由器中已经处理了哪些消息。当路由器 x 启动一个广播消息时，给它赋予一个唯一序列号 (sequence number) k 。因此，数对 (x, k) 标志一个洪泛消息。当路由器接收到带有 (x, k) 的洪泛消息时，则检查 (x, k) 是否在它自己的表中。如果在表中，则简单地抛弃该消息。否则，把 (x, k) 添加到表中，重新把消息广播给它的近邻。

这种方法肯定可以消除无限循环问题，但它不是空间有效的算法。普遍使用的启发式策略是使每个路由器只保存网络中每个其他路由器的最近序列号，且假设如果路由器接收到 x 发起的序列号为 k 的消息，则它可能已经处理了 x 发送的且序列号小于 k 的所有消息。

3. 洪泛算法分析

为了分析洪泛算法，首先注意到洪泛算法没有设置代价，但假设每个路由器知道网络中的路由器数 n 和网络的直径 D 。如果利用跳数计数器的启发式策略，那么路由器中不需要额外的空间。而如果利用序列号启发式策略，那么最坏情况下每个路由器所需空间为 $O(n)$ 。在这两种情况下，路由器处理一条消息所需要的期望时间与路由器的近邻数成正比，这是因为在散列表中进行查找和插入的期望运行时间为 $O(1)$ (见2.5.3节)。

现在分析洪泛算法的消息复杂度。当利用跳数计数器的启发式策略时，在最坏情况下，消息复杂度为 $O((d_{\max}-1)^D)$ ，其中 d_{\max} 是网络中路由器的最大度。当利用序列号的启发式策略时，消息复杂度为 $O(m)$ ，因为我们以沿着网络中的每条边发送一条消息结束。因为 m 通常比 $(d_{\max}-1)^D$ 小得多，所以序列号法一般比跳数计数器法更受欢迎。无论哪一种情况，洪泛算法仅对那些希望广播给网络中所有其他路由器的消息才是有效的。

在同步网络模型中，洪泛算法仍然保证把一条消息 M 从它的源点发送到每个目的点所用的跳数最少，即它总会找到一条最短路由路径。给定与洪泛算法相关的代价，最好使算法能够沿着最短路径更有效地路由消息。我们下面讨论的算法正好做到了这一点，它通过进行一些设置计算，确定网络中良好的路由。

531

11.3.2 单播路由的距离矢量算法

我们讨论的第一个单播路由算法是Bellman和Ford提出的用于在图中找出一条最短路径的经典算法 (7.1.2节) 的一个分布式版本。假设网络中的每条边 e 上有一个正权值 $w(e)$ ，表示发送消息通过 e 的代价。例如，权值可以表示通信链路的平均“延迟”。

单播路由问题是建立网络中结点的数据结构，支持消息从源点到目的点的有效路由。

距离矢量 (distance vector) 算法总是沿着最短路径路由。算法的主要思想是对于每个路由器 x ，存储一个桶数组，称它为路由器 x 的距离矢量。它存储从 x 到网络中其他每个路由器 y 的已知的最佳路径长度，用 $D_x[y]$ 表示。这个路径上的第一条边 (连接) 用 $C_x[y]$ 表示。起初，每条边赋值为

$$D_x[y] = D_y[x] = w(x, y)$$

且

$$C_x[y] = C_y[x] = (x, y)$$

其他所有 D_x 项均设置为等于 $+\infty$ 。然后执行一系列的轮。反复迭代使每条距离矢量逐步求精,找出可能的更好路径,直到每个距离矢量存储距所有其他路由器的实际距离。

1. 分布松弛

距离矢量算法设置由一系列轮组成。反过来,每一轮都由松弛(relaxation)步骤集合组成。在每一轮开始,每个路由器把它的距离矢量发送给网络中它的所有直接近邻。在路由器 x 收到来自它的每个近邻的当前距离矢量后,执行以下局部计算 $n-1$ 轮:

```
for 每个与 $x$ 相连的路由器 $w$  do
  for 网络中的每个路由器 $y$  do
    {松弛}
    if  $D_w[y] + w(x, w) < D_x[y]$  then
      {找到通过 $w$ 的从 $x$ 到 $y$ 的一条更好的路由}
       $D_x[y] \leftarrow D_w[y] + w(x, w)$ 
       $C_x[y] \leftarrow (x, w)$ 
```

532

2. 性能

在每一轮中,因为每个顶点 x 向它的每个近邻发送大小为 n 的一个矢量, x 完成一轮的时间和消息复杂度为 $O(d_x n)$,其中 d_x 是 x 的距离, n 是网络中的路由器数量。回忆这个算法迭代 $n-1$ 轮。实际上,可以把它改进到 D 轮,其中 D 是网络的直径。因为在许多轮之后,距离矢量不会改变。

设置之后,路由器 x 的度矢量存储 x 到网络中其他路由器的实际距离,以及这条路径中的第一条边。于是,一旦执行这种设置,路由算法相当简单:如果路由器 x 接收到打算发送给路由器 y 的消息,那么 x 沿着边 $C_x[y]$ 向 y 发送这条消息。

3. 距离矢量算法的正确性

距离矢量算法的正确性由一个简单的归纳证明而得。

引理11.1 在第 i 轮结束时,每个距离矢量存储到达其他每个路由器的最短路径,并限制沿着这条路径至多访问 i 个其他的路由器。

这个事实在算法开始时为真,每一轮中所作的松弛保证每一轮后也为真。

4. 距离矢量算法分析

我们分析距离矢量算法的复杂度。与11.3.1节中讨论的洪泛算法不同的是,距离矢量算法有大量设置代价,每一轮所传递的消息总数与网络中所有度之和的 n 倍成正比。这是因为路由器 x 发送和接收的消息数是 $O(d_x n)$,其中 d_x 表示 x 的度。因此,每一轮中有 $O(mn)$ 条消息。于是,距离矢量算法设置的消息总复杂度是 $O(Dmn)$,在最坏情况下它是 $O(n^2 m)$ 。

设置完成之后,每个路由器把 $n-1$ 个元素存储在桶数组中,占用 $O(n)$ 的空间。处理一条消息的期望运行时间为 $O(1)$ 。注意所需局部空间与带有序列号的洪泛算法的空间一样。因为距离矢量算法像洪泛算法那样找出最短路径,但在设置计算的过程中进行。它可看成用设置代价换取后面有效的消息发送。

533

11.3.3 单播路由的链路-状态算法

最后讨论的静态路由算法是Dijkstra提出的经典的最短路径算法的一个分布式版本。正如在距离矢量算法中所描述的那样,假设边上具有正权值。距离矢量算法在一系列轮中执行设置过程,

每一轮中只要求相邻路由器之间局部通信，而链路-状态算法计算单个通信轮，要求贯穿整个网络的大量全局通信。

链路-状态算法 (link-state algorithm) 的工作过程如下。利用带有序列号的洪泛路由算法 (不要求预先设置)，每一个路由器 x 把它所依附边上 (即 x 的“状态”) 的权值广播给网络中的所有其他的路由器。广播阶段之后，每个路由器都知道整个网络，并且可以运行 Dijkstra 最短路径算法，确定从这个路由器到其他每个路由器 y 的最短路径，以及这条路径上的第一条边 $C_x[y]$ 。利用标准 Dijkstra 算法实现这个内部计算需要 $O(m \log n)$ 的时间，或利用更复杂的数据结构 (回忆 7.1.1 节) 实现这个计算则需要 $O(n \log n + m)$ 的时间。

正如在距离矢量算法中的那样，每个路由器上设置过程构造的数据结构占据 $O(n)$ 的空间，处理消息的期望时间为 $O(1)$ 。

现在分析链路-状态算法中的设置的消息总复杂度。总共需要广播 m 条常数大小的消息，每条消息引起洪泛算法依次发送 m 条消息。因此，设置的消息总复杂度为 $O(m^2)$ 。

广播路由算法和单播路由算法的比较

表 11-1 中比较了本节中讨论的三个静态路由算法的性能。注意这里没有包括路由器在预处理设置计算中所需的内部计算时间。

表 11-1 静态路由算法的渐近性能界限。利用网络和路由参数的下列符号: n , 结点数; m , 边数; d , 最大结点数; D , 直径; p , 最短路由路径中的边数。注意 p 、 d 和 D 都小于 n

算法	消息	局部空间	局部时间	路由时间
洪泛 w/跳数	$O(1)$	$O(1)$	$O(d)$	$O((d-1)^p)$
洪泛 w/序列号	$O(1)$	$O(n)$	$O(d)$	$O(m)$
距离矢量	$O(Dnm)$	$O(n)$	$O(1)$	$O(p)$
链路状态	$O(m^2)$	$O(n)$	$O(1)$	$O(p)$

534

11.4 多播路由

到目前为止，已经研究了两种通信类型的路由算法——广播路由算法和单播路由算法——这是两种极端情况。介于中间的是多播路由 (multicast routing)，它涉及与网络中主机的一个子集进行通信，这个子集称为多播组 (multicast group)。

11.4.1 逆向路径转发

逆向路径转发 (RPF) 算法将用于广播路由的洪泛算法改编成用于多播路由。它利用每个路由器上可用的现有最短路径路由表工作，沿着从源点开始的最短路径树广播一条多播消息。

方法开始于一台希望把一条消息发送给组 g 的主机。主机把那条消息发送给它的局部路由器 s ，然后 s 把这条消息发送给它的所有近邻路由器。当某个路由器 x 接收到来自其中一个近邻 y 的源于路由器 s 的多播消息时， x 检查它的局部路由表，查看 y 是否是在从 x 到 s 的最短路径上。如果 y 不在从 x 到 s 的最短路径上，那么 (假设所有路由表都是正确的和一致的) 从 y 到 x 的链路不在源为 s 的最短路径上。因此，在这种情况下， x 简单地抛弃来自 y 的包，并向 y 发回一个特殊的修剪 (prune) 消息，该消息包括源 s 的名字和组 g 的名字。这条修剪消息告诉 y 停止发送组 g 想要的来自 s 的多播消息 (且如果可能，不再沿着这条链路发送的来自 s 的任何多播消息，而不管这个组是哪一个组)。另一方面，如果 y 在从 x 到 s 的最短路径上，那么 x 复制这条消息，并把它发送给它的所有近邻路由

器,除了路由器 y 自身之外。因为在这种情况下,从 y 到 x 的链路在源为 s 的最短路径上(再次假设所有路由表是正确的和一致的)。

这种广播通信模式从 s 沿着最短路径树 T 向外扩展,直到把消息洪泛到整个网络。事实是如果网络中一小部分路由器有客户希望接收到发送给组 g 的多播信息,那么算法把一条多播消息广播给网络中的每个路由器就是一种浪费。

535

为了处理这种浪费,RPF算法提供另一种类型的消息修剪。尤其是,如果路由器 x 在 T 中的一个外部结点上,确定在它的局域网中没有客户对接收组 g 的消息感兴趣,那么 x 向它在 T 中的父结点 y 发出修剪消息,告诉 y 停止给它发送从 s 到组 g 的消息。这条消息实际上告诉 y 从树 T 中删除外部结点 x 。

1. 修剪

当然这些修剪消息可被 T 中多个外部结点并行发出;因此,同时会删除 T 中的多个外部结点。此外,删除 T 中的某些外部结点会创建新的外部结点。因此,RPF算法不断测试每个路由器 x 是否变成 T 中的一个外部结点。如果 x 变成一个外部结点,那么它应该测试它的局域网中是否有客户主机对接收组 g 的多播消息感兴趣。再次,如果没有这样的客户,那么 x 向它在 T 中的父结点发送从 s 到组 g 的修剪消息作为多播消息。这个外部结点的修剪过程会一直继续,直到 T 中所有剩余的外部结点有客户希望接收到组 g 的多播消息。此时,RPF算法达到稳定状态。

然而这个稳定状态不能永远被锁定,因为网络中可能有一些客户主机希望开始接收到组 g 的消息,即至少可能有一个客户 h 希望加入组 g 中。因为RPF没有为客户加入(join)一个组提供明确的方式。 h 能够开始接收到组 g 的多播消息的唯一方式是 h 的路由器 x 正在接收这些包。但如果 x 已经从 T 中被修剪掉,那么这种情况不会发生。因此,RPF算法的另一部分是在一段时间之后,在因特网的一个结点中存储的修剪会超时。

说从路由器 z 到路由器 x 的修剪保存 z 的前一条修剪消息,当出现超时,将重新向 z 发送到组 g 的多播包。因此,如果一个路由器确实不希望接收或处理某一种类型的多播消息,那么它必定会不断地利用修剪消息,把它的这个愿望通知它上游近邻。由于这个原因,RPF算法不是一种进行多播路由的有效方法。

2. 性能

就消息的效率而言,RPF起初发出 $O(m)$ 条消息,其中 m 是网络中的连接(边)数。但是,在第一次的修剪波通过网络传播后,每次多播都会使多播的复杂度减少到 $O(n)$ 条消息,其中 n 是路由器数量。随着更多的修剪消息被处理,这个数量还可以进一步减少。就路由器中的额外存储量而言,RPF算法不太有效,因为它要求每个路由器存储接收的每条修剪消息,直到修剪消息超时。然而在最坏情况下,路由器 x 可能存储多达 $O(|S| \cdot |g| \cdot d_x)$ 条修剪消息,其中 S 是源点的集合, g 是组的集合, d_x 是 x 在网络中的度。因此,正如已经观察到的那样,RPF算法不是有效算法。

536

11.4.2 中心树

比逆向路径转发算法效率更高的多播算法是中心树(center-based tree)方法。在这个算法中,对于每个组 g ,选择网络中的一个路由器 z ,作为组 g 的“中心”结点或“集中”结点。路由器 z 形成 g 的多播树 T 的根结点,这个结点用于向组中的路由器发送消息。任何源点想要向组 g 中发送一条多播消息,首先都会把那条消息发到中心 z ,在中心树算法的最简单的形式中,这条消息始终会发送到 z ,一旦 z 接收到这条消息,那么就会把它广播给 T 的所有结点。因此, T 中一个结点表示的每个路由器 x 都知道,如果 x 从它在 T 中的父结点接收到 g 的多播消息,那么 x 复制这条消息,并把它发送给它的所有近邻,这些近邻就是它在 T 中的子结点。同样,在中心树方法的最简单的形式中,如果 x 接收到来自除它在 T 中父结点之外的其他任何近邻的一条多播消息,那么它

把这条消息沿树向上发送给 z 。这样一条消息来自某个源点，应该在多播到 T 之前将其发送给 z 。

如上所述，我们必须显式维持每个组 g 的多播树。因此，必须为路由器提供一种加入（join）到组 g 中的方式。路由器 x 的加入操作首先是向中心结点 z 发送一条加入消息。接收到来自近邻 t 的一条加入消息的任何其他路由器 y 会查找路由表，以查看它的哪个近邻 u 在到 z 的最短路径上，然后 y 创建和存储一条内部记录，表明 y 现在在树 T 中有子结点 t 和父结点 u 。如果 y 已经在树 T 中（即已有一条记录说明 u 是 y 在 T 中的父结点），那么这就完成了加入操作——路由器 x 现在是树 T 中的一个连通外部结点。如果 y 尚未在树 T 中，那么 y 把加入消息向上传播给它在 T 中的（新）父结点 u （见图11-6）。

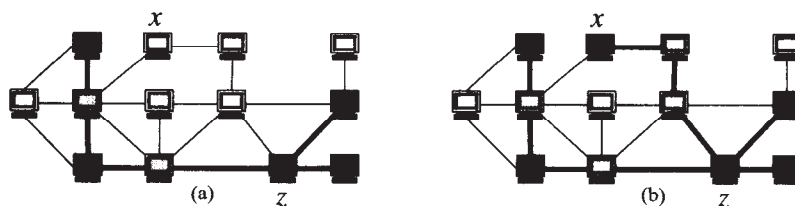


图11-6 中心树协议的说明。(a)中心为 z 的多播树，其中粗边是树边，深灰结点是组成员，浅灰结点是简单传播消息的路由器；(b) x 加入后的多播树。注意只用一条边可将 x 加入，且没有中间结点，但到 z 的路由可能不是最短路径

537

性能

对于希望离开组 g 的路由器（要么通过显式离开（leave）消息，要么由于加入记录超时），将会执行加入操作的逆过程。在效率方面，中心树方法只沿着组 g 的多播树 T 发送消息，因而每次多播的消息复杂度为 $O(|T|)$ 。此外，因为每个路由器只存储 T 的局部结构，任何一个路由器上需要存储记录的最大数量是 $O(|g|d_x)$ ，其中 g 是组集合 d_x 是 x 在网络中的度。于是，中心树算法的效率上的好处来自总是沿着一棵树 T 发送组 g 的多播消息。然而，对于发送这样的消息，这棵树可能不是最有效的树。

11.4.3 Steiner 树

Steiner树算法的主要目标是优化所有边上的总代价，这些边用于构建组 g 的多播树。形式上，设 $G=(V, E)$ 表示我们的网络，其中 V 表示路由器集合，定义 G 中边上的代价度量为 $c(e)$ 。此外，假设给定 V 的一个子集 S ，表示网络中属于组 g 的路由器。 S 的Steiner树（Steiner tree） T 是连接 S 中所有结点且总代价最小的树。注意如果 $|S|=1$ ，那么 T 是单个结点；如果 $|S|=2$ ，那么 T 是 S 中两个结点的最短路径；如果 $|S|=|V|$ ，那么 T 是 T 的一棵最小生成树。因此，对于这些特例，可以容易地构造连接 S 中结点的最优树 T 。可以利用这棵树进行到组 g 的所有多播，如在中心树算法中那样。不幸的是，求解Steiner树问题的一般情况是NP难题（13.2.1节）；因此，我们应该试图利用某些启发式搜索算法求Steiner树问题的近似算法。

我们描述的Steiner树近似算法基于上述的最小生成树算法（11.2.4节）。它利用了网络中每个路由器上的可用信息，称为距离网络（distance network）算法，因为它工作在 G 的导出图 G' 上，称 G' 为 S 的距离网络。取 S 中的顶点作为 G' 中的结点，并用边连接每对这样的顶点构成 G' 。 G' 中边 $e=(v, w)$ 上的代价 $c'(e)$ 正是 G 中 v 和 w 之间的距离，即 G 中从 v 到 w 的最短路径长度（这里假设 G 是无向图）。算法如下：

- (1) 构造 S 的距离网络 G' 的一种表示法。
- (2) 找出 G' 中的一棵最小生成树(MST) T_M 。

538

(3) 对于 T_M 中的每条边 (v, w) , 取从 v 到 w 的最短路径, 把 T_M 转换成 G 中的一棵树 T_A 。

(4) 找出 T_A 中结点导出的 G 的子图的最小生成树 T_D 。

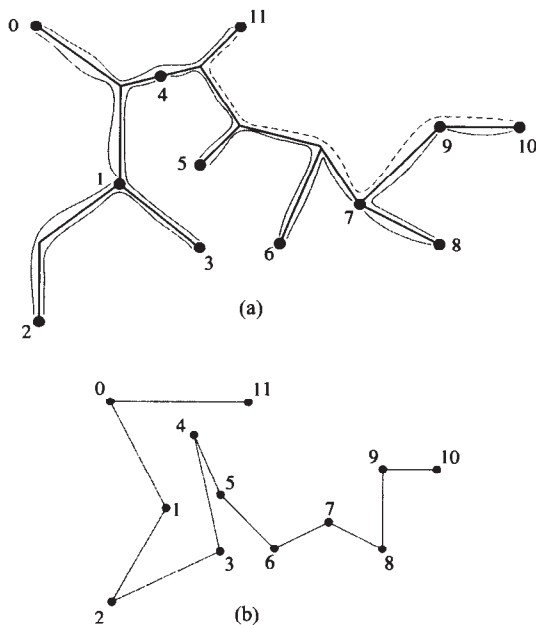
(5) 从 T_D 中删除导致不在 S 中的外部结点的任何路径。

最后两步是对树 T_A 的一种改进, 它已是Steiner树 T_S 的一种很好的近似。

引理11.2 $|T_A| \leq |T_S|(2 - 2/k)$, 其中 $k = |S|$ 。

证明 考虑Steiner树 T_S , 它把 S 中的结点优化地连接起来。进一步考虑行进路线 W , 它由沿着 T_S 边界周围连续遍历而得。也就是说, 想象 T_S 是某个道路集合的一张图, 我们总是沿着道路的右侧行走。因此, $|W| = 2|T_S|$ 。把 W 划分成 k 个路径的集合, 这些路径把 S 中的结点对连接起来, 可通过遍历 W 连续访问这些路径。仅在第一次访问每个结点时划分 W (见图11-7a)。设 P 表示从这 k 条路径中删除代价最高的路径后剩下的 $k-1$ 条路径的集合。因此, $|P| \leq (2 - 2/k)|T_S|$ 。设 P' 表示用 S 中从 v 到 w 的最短路径替换 P 中的每条路径 p 所形成的路径集合, p 表示 S 中从结点 v 到 w 的路径。因此, $|P'| \leq |P|$ 。最后, 设 T' 表示距离网络中的边集合, 用 S 的距离网络中的相应边替换 P' 中的每条路径得到该距离网络 (见图11-7b)。观察可见 T' 是 S 的距离网络中的一棵生成树; 因此, $|T_A| \leq |T'|$ 。于是 $|T_A| \leq (2 - 2/k)|T_S|$ 。 ■

因为 T_D 不会比 T_A 更坏, 上述引理蕴涵着 $|T_D| \leq (2 - 2/k)|T_S|$ 。



539

图11-7 引理11.2证明的说明: (a)Steiner树 T_S , 其中 S 中的顶点突出显示, 把行进路线 W 划分成 $k = 12$ 条路径。最高代价的路径用虚线表示; (b)距离网络中的树 T'

分析

距离网络算法除了产生Steiner树的一种良好的表示法之外, 还容易在网络环境中实现。距离网络自身可以间接地用距离矢量表示, 它可以存储在网络中的每个路由器中 (链路-状态算法或距离-矢量算法)。因此, 如果每个路由器知道那个路由器属于组 g , 那么无需额外的计算即可构造距离网络 G' 。可利用上述11.2.4节描述的分布式最小生成树算法找出 G' 中的最小生成树, 这里考虑 S 中每个结点到所有其他结点的最短路径, 而不是只检查一轮中相邻连接的结点。但是如果

每个结点都存储到达网络中所有其他结点的距离矢量,那么不需要查询其他结点,就可以确定到其他结点的距离。在每一轮中,只需要确定 S 中每个结点 x 到 S 中与 x 不同连通分量中的另一结点的最短距离,即可以通过路由 G 中的消息执行分布式算法的每一步。因为每个这样的消息至多要求 D 个跳数,这蕴涵着可以利用至多 $O(k^2D)$ 条消息找到 T_k ,其中 D 是 G 的直径。如果还想要计算 T_D ,那么必须另外执行 G 自身中的MST算法,这可用 $O(n \log n + m)$ 条消息完成。因此,这个Steiner树问题的近似算法具有有效的消息复杂度。

然而,这个算法只适合于那些随着时间的推移而相当稳定的组,因为组的任何改变都会要求重新建立那棵树的一棵新的多播树。注意,我们讨论的每个多播算法都有缺点。的确,这仍然是一件值得考虑的算法上的工作,可能提出新的更有效的多播路由算法。

540

11.5 习题

基础题

- R-11.1 画图说明10个结点的环网上同步领导人选举算法每一轮中发送的消息。
- R-11.2 画图说明10个结点的树上同步领导人选举算法每一轮中发送的消息。设计一个过程,使得两个结点“平局”选作领导人。
- R-11.3 画图说明同步BFS算法的工作过程。你的图中应该包括一棵至少5层的BFS树。
- R-11.4 画图说明异步BFS算法的工作过程。你的图中应该包括一棵至少3层的BFS树,并且应该同时显示向下脉冲行为和向上脉冲行为。
- R-11.5 画出一个计算机网络,使同步分布式Baruvka最小生成树算法运行4轮。显示每一轮中选择的边。
- R-11.6 利用跳数计数器,给出计算机执行路由洪泛消息的算法的伪代码描述。
- R-11.7 利用序列号,给出计算机执行路由洪泛消息的算法的伪代码描述。
- R-11.8 给出在计算机执行了链路状态算法或距离矢量算法的设置之后,路由一条单播消息的算法的伪代码描述。
- R-11.9 画图说明距离矢量算法如何在至少有10个结点的网络中工作。你的说明应该显示至少经过三轮才稳定的计算。
- R-11.10 在具有10个结点、20条边的网络中,执行链路状态设置算法正好要发送多少条消息?假设每个结点已经知道它所依附边的状态。
- R-11.11 画图说明逆向路径转发多播算法中的三种修剪操作。
- R-11.12 画图说明多播路由的中心树方法,它可以使用的边数是Steiner树方法的两倍。
- R-11.13 画图说明多播路由的逆向路径转发方法,它可以使用的边数多于Steiner树方法。

541

创新题

- C-11.1 扩展领导人选举算法,使之适合网络是无向回路且消息可在任一方向上传递的情况。
提示:把无向回路看作两个有向回路的并集。
- C-11.2 考虑 $n \geq 4$ 个处理器的环网中选举领导人处理器的问题,每个处理器有唯一id。假设所有处理器都知道 n 的值,设计一个同步随机化算法,它的期望运行时间为 $O(n)$,期望的消息复杂度也是 $O(n)$ 。
提示:利用如下事实,即如果 $n \geq 4$ 个处理器中的每一个以 $4/n$ 的概率独立决定发送一条消息,那么发送消息的期望处理器数量为4,没有处理器发送消息的概率小于 $1/2$ 。
- C-11.3 考虑在 n 个处理器的环网中选举领导人处理器的问题,每个处理器有唯一id。设计一个同步确定性算法,它的运行时间和消息复杂度均为 $O(n \log n)$ 。
提示:考虑有 $\log n$ 个阶段的算法,在第 i 阶段,每个处理器认为它可能是领导人,在每个方向发出一条“探测”消息,经过 2^i 个跳数,如果它发现没有处理器具有更低的id值,则返回。
- C-11.4 为具有 n 个顶点、 m 条边的网络设计一个异步广度优先查找(BFS)算法,它的消息总复杂度为 $O(nh)$

+ m), 其中 h 是 BFS 树的高度。

提示: 扩展“向下脉冲”和“向上脉冲”消息, 使得根 s 知道构造完整 BFS 树的准确轮数。

- C-11.5 假定一个连通网络 G 不含回路; 即 G 是一棵(自由)树。描述对 G 中结点计数的一种方法, 使得结点 x 只存储每条依附边的 $O(1)$ 条信息, 但可以不经绕道地把一条消息从 G 中的计算机 y 路由到计算机 z 。

- C-11.6 描述一个同步分布式算法, 用于找出具有 n 个顶点、 m 条边的加权网络中的一棵最小生成树, 要求消息复杂度为 $O(m + n \log n)$ 。

提示: 考虑首先对依附每个顶点的边进行排序的优点。

- C-11.7 描述一个异步分布式算法, 找出具有 n 个顶点、 m 条边的加权网络中的一棵最小生成树, 要求消息复杂度为 $O(m \log n)$ 。

提示: 考虑利用“脉冲”策略对算法执行的轮数进行计数。

- C-11.8 描述如何修改 11.2.4 节描述的分布式最小生成树算法, 使得无需假设边上的权值是唯一的。

542

提示: 描述一个局部打破平局的规则, 使得在这个规则之下的最小生成树是唯一的。

- C-11.9 考虑一个利用序列号的洪泛算法的被动动态版本, 使得算法运行时可以应对出现网络分割的情况。该方法的主要思想是, 使发起洪泛广播的每个路由器存储旧消息的时间比预期网络故障时间要长。每个路由器仍然存储来自其他每个路由器的洪泛消息的最新序列号。但现在如果一个路由器 x 接收序列号为 i 的洪泛消息 y , 该序列号要比 x 以前记录的来自 y 的序列号 j 大 1, 那么 x 发起一条应答洪泛消息, 请求 y 通过 $i-1$, 重发它的消息 $j+1$ 。表明如何调整这个可能导致无限循环的洪泛消息。

- C-11.10 考虑一个利用序列号或跳数计数器的洪泛算法版本, 每当路由器 x 不能向近邻路由器 y 转发一条消息 M 时(例如, 由于连接 (x, y) 关闭), 那么 x 把这条消息放在缓冲队列中。你的算法应该工作在动态可调的情况下, 其中通信链路可能发生故障和恢复。设计一个协议, 使得当链路 (x, y) 恢复时, x 向 y 发送以前队列中关于这个连接的所有消息。证明即使在网络暂时分割的情况下, 也允许洪泛消息健壮地发送。实现这个方案需要多大的附加空间? 如何修改这个算法, 使计算机永远与网络断开连接。

- C-11.11 设 G 是表示为图的网络, 图中 n 个顶点表示路由器, m 条边表示连接。进一步假定 G 是静态的(即网络不变化), 且存在定义在 G 中结点上的—棵生成树 T , T 不必是一棵最小生成树, 但包括 G 中所有顶点, 并对于 G 中的每个结点 v , v 存储它在 G 中的近邻, 它还知道这些近邻中有哪些也是它在 T 中的近邻。描述如何利用树 T , 改进执行链路状态最短路径的设置算法的消息复杂度(构建最短路径路由的路由表)。改进后的算法的消息复杂度是多少?

- C-11.12 假定一个路由算法(如链路状态算法或距离矢量算法)在静态网络(链路没有变化)中完成了它的设置阶段。也就是说, 每个路由器存储一个向量 D , 满足 $D[j]$ 存储从路由器 i 到路由器 j 的距离, 以及到 j 的路径上下一个跳的名字。设计一个有效算法, 允许所有路由器验证它们的路由表是正确的。这个算法的消息复杂度是多少?

- C-11.13 假定利用中心树方法实现一个多播算法, 使得多播组中的每个成员通过向树根发送一条加入消息, 动态地进入小组, 这条消息遍历的路径形成了中心树中的一条新路径(联合中心树中某些以前的结点)。成员离开这棵树的过程是类似的, 但按逆向方式。描述如何修改加入算法, 使得每次某个路由器 x 想要加入小组时, 能够自动地连接到中心树中与 x 最近的结点(注意这个结点实际上可能不是组成员)。

543

- C-11.14 证明如何对上一个问题的解决方法进行扩展, 使得不是找出到新路由器 x 的最近树结点, 而是找出到 x 的最近组成员。这个结点是进行任意播(anycast)的理想候选结点, 它要求某个组中的任何成员对某个查询做出响应。

- C-11.15 假定在标准洪泛算法(利用序列号)中利用数字签名, 使得想要发送洪泛消息的每个路由器必须签名那条消息。在传输过程中, 如果任何路由器接收到不是由源路由器正确签名的包, 那么就会丢弃这个包。算法仍然允许源 x 签名具有相同序列号的两条不同的洪泛消息, 并同时把它们发送给不同的近邻。描述一个算法, 用于检测是否会出现这样的问题, 假设网络是双连通的, x 是唯一的“坏”路由器。你所设计算法的消息复杂度是多少?

程序设计

P-11.1 利用网络访问协议（如“ping”或“HTTP”）收集因特网上4台主机的以下统计信息。这4台主机在物理位置上分布在地球上4个不同的洲（即下列7个洲中的4个洲：北美洲、南美洲、欧洲、亚洲、非洲、澳洲和南极洲）：

- 50个连续包请求所花费的最短时间、平均时间、标准方差和最大来回旅程时间（这些请求相互之间都发生在几秒钟之内）。把数据画成分散点图，其中 x 轴表示每个来回旅程， y 表示来回旅程时间。
- 10个连续包请求所花费的平均来回旅程时间（这些请求相互之间都发生在几秒钟之内），但至少每4个小时重复一次，持续至少两天。理想情况下，应该重复每小时ping 10次的实验，持续5天（利用后台进程）。把数据画成图，其中 x 轴表示天时， y 轴表示来回旅程时间。在同一幅图上层叠所有4台主机的统计直线图。

P-11.2 做一个实验，对Steiner树多播方法和中心树方法的近似算法进行比较。哪种方法对边使用要好很多？如果这样，则对连接请求的分布做了什么假设？

11.6 本章注记

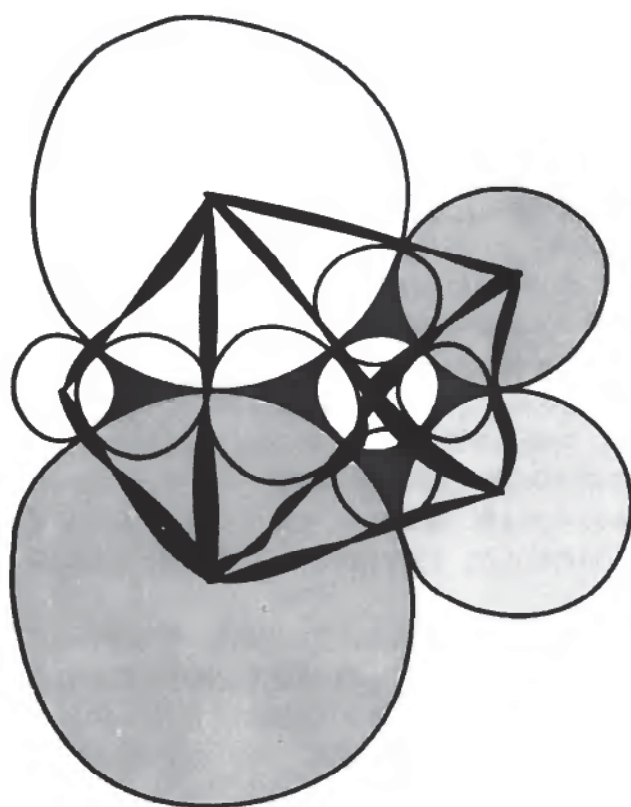
对分布式算法的进一步研究感兴趣的读者，可以参考Lynch[137]、Peleg[166]和Tel[202]的优秀著作。对计算机网络和协议感兴趣的读者可进一步阅读Comer[52]、Huitema[105]和Tanenbaum[196]的著作。Barůvka算法的分布式实现由Gallego等人[74]提出。在异步网络中实现它的一种简单方法由Awerbuch[18]给出。

1

Part 4

第四部分

其他主题





我们生活在一个多维的几何世界里。物理空间自身是三维的，因为我们可以利用三维坐标 x 、 y 和 z 描述空间中的点。完整地描述一个机械手的指尖实际上需要六维，因为利用三维描述指尖的空间位置，以及用另外三维描述指尖的角度（一般称为俯仰角、倾斜角和偏航角）。描述飞行中的飞机状态至少需要九维，因为需要六维描述它的方位，这与描述机械手的指尖一样，还需要三个以上的维描述飞机的速度。事实上，这些物理表示可认为是“低维的”，尤其是在机器学习或计算生物学的应用中，100维和1000维的空间也不足为奇。本章以计算几何（computational geometry）为导向，研究处理几何数据（如点、线和多边形）的数据结构和算法。

实际上有大量处理多维几何数据的不同数据结构和算法。讨论所有这些超出了本章的范围。代之以在本章只对一些更有趣的方法做一些引导性介绍。首先讨论范围树（range tree），它可存储多维数据点，支持一种特殊的查找操作，称为范围查找查询（range-searching query），还包括范围树的另一种有趣的变体，称为优先查找树（priority search tree）。最后，讨论一类称为划分树（partition tree）的数据结构，它把空间分成单元，我们集中讨论称为四叉树和 k -d树的变体。

我们接着介绍一种称为平面扫描（plane-sweep）技术的通用算法设计方法。这种技术在解决二维几何问题中非常有用。通过利用这种技术把二维问题归约到一些一维问题，通过求解二维几何问题来说明这种技术的应用。确切地讲，把平面扫描技术应用于正交线段交集的计算，找出一个点集中具有最小距离的点对。平面扫描技术还可用于求解其他问题，有些问题将在本章后面的习题中探讨。

本章最后研究的一个问题可应用到计算机图形学、统计学、地理信息系统、机器人和计算机视觉学中。在这个称为凸包（convex hull）问题的应用中，我们对找出某一给定点集的最小凸包集合感兴趣。构造这样一个集合，允许定义点集的“边界”；因此，这个问题也引出一些关于如何表示几何对象和进行几何测试的有趣问题。

12.1 范围树

多维数据出现在各种应用中，包括统计学和机器人的应用中。多维数据的最简单类型是 d -维点，可用数值坐标（coordinate）的一个序列 $(x_0, x_1, \dots, x_{d-1})$ 表示。在商务应用中，一个 d -维点可以表示数据库中一个产品或者雇员的各种属性。例如，电子类的电视有不同的属性值，它们是价格、屏幕大小、重量、高度、宽度和厚度。多维应用数据也可以来自于科学应用，每个点表示某个实验或观察的属性。例如，天文学太空测量中的天体可能有不同的属性值，如亮度（或外观

大小)、直径、距离和太空中的位置(它自身是二维的)。

对多维点集的自然查询操作是范围查找查询(range-search query),这个查询要求检索坐标位于给定区域中的多维集合中的所有点。例如,一位顾客想要买一台新电视机,按照电子商店分类,可能在查询所有屏幕大小介于24英寸~27英寸之间,且价格在200美元~400美元之间的电视机。而对研究行星感兴趣的天文学家可能要求查询距离介于1.5~10个天文单位之间、外观大小介于+1~+15之间以及直径介于0.5~1000公里之间的太空中的所有物体。本节中讨论的范围树的数据结构可用于答复这样的查询。

二维范围查找查询

为使讨论简洁,我们集中讨论二维范围查找查询。习题C-12.6指出相应的二维范围树数据结构可被扩展到更高的维。一个二维字典(two-dimensional dictionary)是一个存储关键字-元素数据项的ADT,满足关键字是一个数对 (x, y) ,称为元素的坐标(coordinate)。二维字典 D 支持以下基本查询操作:

- $\text{findAllInRange}(x_1, x_2, y_1, y_2)$: 返回 D 中满足 $x_1 \leq x \leq x_2$ 和 $y_1 \leq y \leq y_2$ 的坐标为 (x, y) 的所有元素。

操作 findAllInRange 是范围查找查询的输出(reporting)版本,因为它要求所有数据项满足范围约束条件。还有一个范围查询的计数(counting)版本,在这个版本中,只是对那个范围中数据项的个数感兴趣。本节其余部分给出实现一个二维字典的数据结构。

549

12.1.1 一维范围查找

在讨论二维查找之前,我们稍微离题一下,首先研究一维范围查找问题。给定一个有序字典 D ,想要进行如下查询操作:

- $\text{findAllInRange}(k_1, k_2)$: 返回 D 中满足 $k_1 \leq k \leq k_2$ 的所有关键字 k 。

我们讨论如何用二叉查找树 T 表示字典 D (第3章),执行查询 $\text{findAllInRange}(k_1, k_2)$ 。利用一个递归方法 1DTreeRangeSearch ,它取范围参数 k_1 和 k_2 以及 T 中的结点 v 作为参数。如果结点 v 是外部结点,则工作完成。如果结点是内部结点,分三种情况,它取决于 $\text{key}(v)$ 的值,即存储在结点 v 处的数据项的关键字:

- $\text{key}(v) < k_1$: 对 v 的右子结点进行递归。
- $k_1 \leq \text{key}(v) \leq k_2$: 输出 $\text{element}(v)$,对 v 的两个子结点进行递归。
- $\text{key}(v) > k_2$: 对 v 的左子结点进行递归。

算法12-1(1DTreeRangeSearch)描述了这个查找过程。图12-1对它进行了说明。通过调用 $\text{1DTreeRangeSearch}(k_1, k_2, T.\text{root}())$,执行操作 $\text{findAllInRange}(k_1, k_2)$ 。直观上讲,方法 1DTreeRangeSearch 是对标准二分搜索方法(算法3-2)修改而成,允许查找两个关键字“ k_1 ”和“ k_2 ”。

算法12-1 二叉查找树中的一维范围查找的递归方法

算法 $\text{1DTreeRangeSearch}(k_1, k_2, v)$

输入: 查找关键字 k_1, k_2 , 以及二叉查找树 T 中的结点 v

输出: 存储在 T 的以 v 为根的子树中的元素, 其关键字大于或等于 k_1 , 并且小于或等于 k_2

if $T.\text{isExternal}(v)$ then

return \emptyset

if $k_1 \leq \text{key}(v) \leq k_2$ then

$L \leftarrow \text{1DTreeRangeSearch}(k_1, k_2, T.\text{leftChild}(v))$

$R \leftarrow \text{1DTreeRangeSearch}(k_1, k_2, T.\text{rightChild}(v))$

return $L \cup \{\text{element}(v)\} \cup R$

```

else if key(v) < k1 then
    return 1DTreeRangeSearch(k1, k2, T.rightChild(v))
else if k2 < key(v) then
    return 1DTreeRangeSearch(k1, k2, T.leftChild(v))

```

550

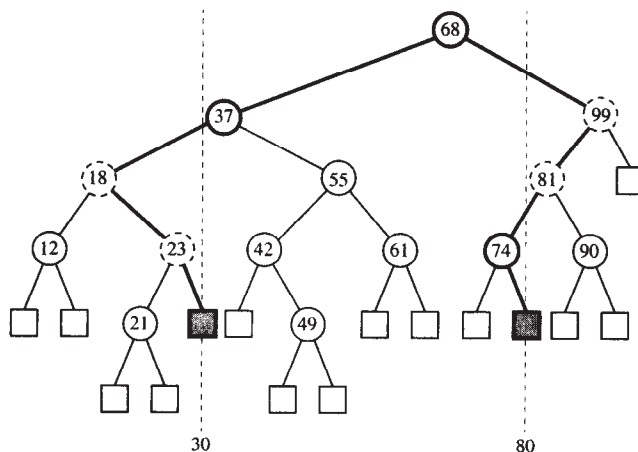


图12-1 利用二叉查找树进行一维范围查找，其中 $k_1 = 30$ ， $k_2 = 80$ 。边界结点的路径 P_1 和 P_2 用粗线表示。其关键字在区间 $[k_1, k_2]$ 之外的存储数据项的边界结点用虚线画出。有10个内结点

性能

现在分析算法1DtreeRangeSearch的运行时间。在分析中，为简单起见，假设 T 中不含关键字为 k_1 或 k_2 的数据项。扩展到一般情况的分析留作习题。

设 P_1 是在树 T 中查找关键字 k_1 时遍历的查找路径。路径 P_1 开始于 T 的根，结束于 T 的外部结点。类似地关于 k_2 定义路径 P_2 。确定 T 中的每个结点 v ，使其属于以下三小组之一：

- 如果结点 v 属于 P_1 或 P_2 ，那么结点 v 是一个边界结点（boundary node）；边界结点存储数据项，其关键字可能位于区间 $[k_1, k_2]$ 之内或之外。
- 如果结点 v 不是边界结点，那么结点 v 是一个内结点（inside node），并且 v 属于根在 P_1 结点右子结点上的子树，或者属于根在 P_2 结点左子结点上的子树；一个内部的内结点存储其关键字位于区间 $[k_1, k_2]$ 内的数据项。
- 如果结点 v 不是边界结点，那么结点 v 是一个外结点（outside node），并且 v 属于根在 P_1 结点左子结点上的子树，或者属于根在 P_2 结点右子结点上的子树；一个内部的外结点存储其关键字位于区间 $[k_1, k_2]$ 外的数据项。

551

考虑算法1DTreeRangeSearch(k_1, k_2, r)的执行，其中 r 是 T 的根。我们遍历边界结点的路径，在左子结点上或右子结点上递归调用算法，直至到达一个外部结点或者到达其关键字在区间 $[k_1, k_2]$ 内的一个内部结点 w （可能是根）。在第一种情况下（到达一个外部结点），算法终止并返回空集。在第二种情况下，算法通过递归调用 w 的两个子结点继续执行。我们知道结点 w 是路径 P_1 和 P_2 的最底部的公共结点。对于每个从这一点访问的边界结点 v ，要么进行对 v 的一个子结点的单一调用，它也是一个边界结点，要么对 v 的一个子结点进行调用且另一个子结点是内结点。一旦访问一个内结点，我们将访问它的所有（内结点）后代。

因为算法访问每个结点花费常数时间，所以算法的运行时间与访问的结点数成正比。如下计算访问的结点数：

- 没有访问外结点。

- 至多访问 $2h+1$ 个边界结点, 其中 h 是 T 的高度, 因为边界结点在查找路径 P_1 和 P_2 上, 它们至少共享一个结点(T 的根)。
- 每次访问一个内结点 v 时, 我们还会访问 T 的以 v 为根的整个子树 T_v , 并把存储在 T_v 中内部结点中的所有元素添加到输出集合中。如果 T_v 中有 s_v 个数据项, 那么它有 $2s_v+1$ 个结点。内结点可被划分成 j 个不相交的子树 T_1, \dots, T_j , 它们以边界结点的子结点为根, 其中 $j \leq 2h$ 。用 s_i 表示存储在树 T_i 中的数据项数, 则访问的内结点总数为

$$\sum_{i=1}^j (2s_i + 1) = 2s + j \leq 2s + 2h$$

于是, 至多会访问 T 中的 $2s + 4h + 1$ 个结点, 操作findAllInRange的运行时间为 $O(h + s)$ 。在最坏情况下, 如果希望最小化 h , 应该选择 T 是一棵平衡二叉查找树, 如AVL树(3.2节)或红黑树(3.3节), 使得 h 为 $O(\log n)$ 。此外, 利用平衡二叉查找树, 额外执行操作InsertItem和removeElement各需要 $O(\log n)$ 的时间。概括如下:

定理12.1 平衡二叉查找树在具有 n 个数据项的有序字典上进行一维范围查找:

- 所用空间为 $O(n)$ 。
- 操作findAllInRange花费的时间为 $O(\log n + s)$, 其中 s 是输出的元素个数。
- 操作InsertItem和removeElement每个花费的时间为 $O(\log n)$ 。

552

12.1.2 二维范围查找

二维范围树(图12-2)是一种实现二维字典ADT的数据结构。它由基于平衡二叉查找树 T 的主(primary)结构和许多辅助(auxiliary)结构组成。正如以下描述的那样, 主结构 T 中的每个内部结点存储对辅助结构的引用。主结构 T 的功能支持基于 x 坐标的查找。为了也能支持对 y 坐标的查找, 利用辅助数据结构的一个集合, 每个数据结构是一棵一维范围树, 利用 y 坐标作为关键字。 T 的主结构是以 x 坐标作为关键字所构建的一棵平衡二叉查找树。 T 的内部结点 v 存储以下数据:

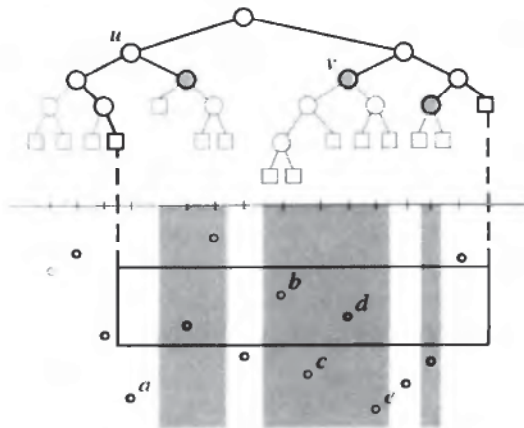


图12-2 二维范围树表示的二维关键字的数据项集合, 以及对它进行的范围查找。主数据结构 T 如图显示。粗线画出的是查找算法访问的 T 中结点。边界结点填充白色, 分配结点填充灰色。存储边界结点 u 的点 a 在查找范围之外。灰色垂直条包含了存储在分配结点的辅助结构中的那些点。例如, 结点 v 的辅助结构中存储了点 b 、 c 、 d 和 e

- 数据项, 用 $x(v)$ 和 $y(v)$ 表示它的坐标, 用element(v)表示它的元素。

553

- 一维范围树 $T(v)$, 它存储的数据项集合与 T 中以 v 为根的子树 (包括 v) 相同, 但利用 y 坐标作为关键字。

引理12.1 存储 n 个数据项的二维范围树所用空间为 $O(n \log n)$, 构造树的时间为 $O(n \log n)$ 。

证明 主结构使用空间为 $O(n)$ 。有 n 个二级结构。辅助结构的大小与存储在其中的数据项数成正比。存储在主结构 T 中结点 v 的数据项也存储在每个辅助结构 $T(u)$ 中, 满足 u 是 v 的祖先。因为树 T 是平衡树, 结点 v 有 $O(\log n)$ 个祖先。因此, 所用总空间为 $O(n \log n)$ 。构造算法留作习题 (C-12.2)。

操作 $\text{findAllInRange}(x_1, x_2, y_1, y_2)$ 的算法首先在主数据结构 T 上进行基本的一维范围查找, 查找范围为 $[x_1, x_2]$, 即向下遍历树 T 查找内结点。然而我们做出一点重要修改: 当到达一个内结点 v 时, 不是递归访问以 v 为根的子树, 而是对 v 的辅助结构进行一维范围查找, 查找范围为 $[y_1, y_2]$ 。

我们称 T 中的内结点为分配结点 (allocation node), 它是边界结点的子结点。算法访问 T 的边界结点和分配结点, 但是不访问其他内结点。算法将每个边界结点 v 分成左结点 (left node)、中间结点 (middle node) 或右结点 (right node)。中间结点是查找 x_1 的路径 P_1 与查找 x_2 的路径 P_2 的交集。左结点在 P_1 中, 但不在 P_2 中。右结点在 P_2 中, 但不在 P_1 中。在每个分配结点 v 处, 算法在辅助结构 $T(v)$ 上执行一维范围查找, 查找范围为 $[y_1, y_2]$ 。

方法的细节如算法12-2所示 (也见图12-2)。

算法12-2 在一棵二维范围树中进行二维范围查找的递归方法。初始方法调用是 $\text{2DTreeRangeSearch}(x_1, x_2, y_1, y_2, T.\text{root}(), \text{"middle"})$ 。算法对所有边界结点关于 x 范围 $[x_1, x_2]$ 进行递归调用。参数 t 表明 v 是一个左边界结点、中间边界结点, 还是右边界结点

```

算法 2DTreeRangeSearch( $x_1, x_2, y_1, y_2, v, t$ ):
    输入: 查找关键字  $x_1, x_2, y_1$  和  $y_2$ ; 二维范围树的主结构  $T$  中的结点  $v$ ; 结点  $v$  的类型  $t$ 
    输出: 以  $v$  为根的子树中的数据项, 它的坐标在  $x$  范围  $[x_1, x_2]$  和  $y$  范围  $[y_1, y_2]$  中
    if  $T.\text{isExternal}(v)$  then
        return  $\emptyset$ 
    if  $x_1 \leq x(v) \leq x_2$  then
        if  $y_1 \leq y(v) \leq y_2$  then
             $M \leftarrow \{\text{element}(v)\}$ 
        else
             $M \leftarrow \emptyset$ 
        if  $t = \text{"left"}$  then
             $L \leftarrow \text{2DTreeRangeSearch}(x_1, x_2, y_1, y_2, T.\text{leftChild}(v), \text{"left"})$ 
             $R \leftarrow \text{1DTreeRangeSearch}(y_1, y_2, T.\text{rightChild}(v))$ 
        else if  $t = \text{"right"}$  then
             $L \leftarrow \text{1DTreeRangeSearch}(y_1, y_2, T.\text{leftChild}(v))$ 
             $R \leftarrow \text{2DTreeRangeSearch}(x_1, x_2, y_1, y_2, T.\text{rightChild}(v), \text{"right"})$ 
        else
             $\{t = \text{"middle"}\}$ 
             $L \leftarrow \text{2DTreeRangeSearch}(x_1, x_2, y_1, y_2, T.\text{leftChild}(v), \text{"left"})$ 
             $R \leftarrow \text{2DTreeRangeSearch}(x_1, x_2, y_1, y_2, T.\text{rightChild}(v), \text{"right"})$ 
    else
         $M \leftarrow \emptyset$ 
        if  $x(v) < x_1$  then
             $L \leftarrow \emptyset$ 
             $R \leftarrow \text{2DTreeRangeSearch}(x_1, x_2, y_1, y_2, T.\text{rightChild}(v), t)$ 

```

```

else
    { $x(v) \geq x_2$ }
     $L \leftarrow \text{2DTreeRangeSearch}(x_1, x_2, y_1, y_2, T.\text{leftChild}(v), t)$ 
     $R \leftarrow \emptyset$ 
    return  $L \cup M \cup R$ 

```

定理12.2 具有 n 个数据项、二维关键字的二维范围树 T 所用空间为 $O(n \log n)$ ，它的构造可用 $O(n \log n)$ 时间完成。利用 T ，二维范围查找查询所用时间为 $O(\log^2 n + s)$ ，其中 s 是输出的元素个数。

证明 由引理12.1可知所用空间和构造时间。现在分析算法12-2 (2DTreeRangeSearch) 的进行范围查找查询的运行时间。对主结构 T 中的每个边界结点和分配结点所花费的时间计数。算法在每个边界结点上所花费的时间为常数。因为有 $O(\log n)$ 个边界结点，边界结点花费的总时间为 $O(\log n)$ 。对于每个分配结点，算法在辅助结构 $T(v)$ 中进行一维范围查找花费的时间为 $O(\log n_v + s_v)$ ，其中 n_v 是存储在 $T(v)$ 中数据项数， s_v 是在 $T(v)$ 中进行范围查找返回的元素个数。设 A 表示分配结点的集合，花费在所有分配结点 A 上的总时间与 $\sum_{v \in A} (\log n_v + s_v)$ 成正比。因为 $|A|$ 为 $O(\log n)$ ， $n_v \leq n$ 且 $\sum_{v \in A} s_v \leq s$ ，则花费在所有分配结点上的总时间为 $O(\log^2 n + s)$ 。由此可得二维范围查找所需的时间为 $O(\log^2 n + s)$ 。 ■

554
555

12.2 优先查找树

在这一节里，提出一种优先查找树结构，可以回答二维关键字的数据项集合 S 的三边范围查询问题：

findAllInRange(x_1, x_2, y_1): 返回 S 中满足 $x_1 \leq x \leq x_2$ 和 $y_1 \leq y$ 的坐标为 (x, y) 的所有数据项。

在几何上，这个查询要求返回两条垂直线 ($x = x_1$ 和 $x = x_2$) 之间以及一条水平线 ($y = y_1$) 上方所有的点。

集合 S 的优先查找树 (priority search tree) 是一棵存储 S 中数据项的二叉树，它的行为就像一棵关于 x 坐标的二叉查找树。为简单起见，假设 S 中的所有数据项的 x 坐标和 y 坐标不同。如果集合 S 为空， T 由一个外部结点组成。否则，设 \bar{p} 是 S 中最顶端的数据项，即 y 坐标最大的数据项。用 \hat{x} 表示 $S - \{\bar{p}\}$ 中数据项的 x 坐标的中值，用 S_L 表示 $S - \{\bar{p}\}$ 中数据项的 x 坐标小于或等于 \hat{x} 的子集，用 S_R 表示 $S - \{\bar{p}\}$ 中数据项的 x 坐标大于 \hat{x} 的子集。递归定义 S 的优先查找树 T 如下：

- 根 T 存储数据项 \bar{p} 和 x 坐标的中值 \hat{x} 。
- T 的左子树是 S_L 的优先查找树。
- T 的右子树是 S_R 的优先查找树。

对于 T 中的每个内部结点 v ，用 $\bar{p}(v)$ 、 $\bar{x}(v)$ 和 $\bar{y}(v)$ 表示存储在 v 中最顶端的数据项及其坐标。同样， \hat{x} 表示存储在 v 中 x 坐标的中值。优先查找树的一个例子如图12-3所示。

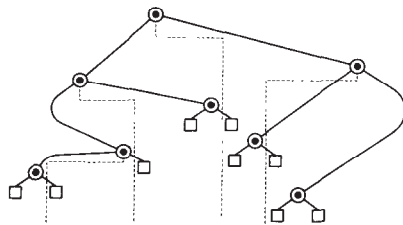


图12-3 二维关键字的数据项集合 S 和 S 的一棵优先查找树 T 。沿着点 $\bar{p}(v)$ 所画的圆圈表示 T 中的每个内部结点 v 。结点 v 下面的虚线表示 x 坐标的中值 $\hat{x}(v)$ ，虚线把存储在 v 的左子树中的数据项与存储在 v 的右子树中的数据项分开

556

12.2.1 构造一棵优先查找树

存储在优先查找树 T 中结点的数据项的 y 坐标满足堆序性质(2.4.3节)。也就是说,如果 u 是 v 的父结点,那么 $\bar{y}(u) > \bar{y}(v)$ 。同样,存储在 T 中结点的 x 坐标的中值定义了一棵二叉查找树(3.1.2节)。这两个事实推动了术语“优先查找树”。于是可以解释从 n 个二维数据项的集合 S 构造一棵优先查找树。先对 S 按照 x 坐标递增的次序排序,然后递归调用算法12-3所示的方法 $\text{buildPST}(S)$ 。

算法12-3 递归构造优先查找树

算法 $\text{buildPST}(S)$:

输入: 按照 x 坐标排序的 n 个二维数据项的序列 S

输出: S 的优先查找树 T

创建一个外部结点 v 组成的优先查找树 T

if $!S.\text{isEmpty}()$ then

 遍历序列 S , 找出 S 中具有最大 y 坐标的数据项 \bar{p}

 从 S 中删除 \bar{p}

$\bar{p}(v) \leftarrow \bar{p}$

$\hat{p} \leftarrow S.\text{elemAtRank}(\lceil S.\text{size}()/2 \rceil)$

$\hat{x}(v) \leftarrow x(\hat{p})$

 把 S 分成两个子序列 S_L 和 S_R , S_L 包含达到 \hat{p} 的数据项(包括 \hat{p}), S_R 包含其余的项

$T_L \leftarrow \text{buildPST}(S_L)$

$T_R \leftarrow \text{buildPST}(S_R)$

$T.\text{expandExternal}(v)$

 用 T_L 替换 v 的左子结点

 用 T_R 替换 v 的右子结点

return T

引理12.2 给定 n 个二维数据项的集合 S , S 的优先查找树利用 $O(n)$ 的空间, 高度为 $O(\log n)$, 构建时间为 $O(n \log n)$ 。

证明 由优先查找树 T 中的每个内部结点存储 S 中的不同数据项, 可得需要 $O(n)$ 的空间。通过将每一层中的结点数分半, 可得树 T 的高度为 $O(\log n)$ 。利用渐近最优的排序算法, 如堆排序或归并排序, 对数据项按照 x 坐标完成排序的时间为 $O(n \log n)$ 。方法 buildPST (算法12-3)的运行时间 $T(n)$ 可用递归方程($T(n) = 2T(n/2) + bn$, 其中常数 $b > 0$)表征。于是, 由主定理(5.3)可知, $T(n)$ 是 $O(n \log n)$ 。 ■

557

12.2.2 优先查找树中的查找

现在显示如何在一棵优先查找树 T 中进行三边的范围查询 $\text{findAllInRange}(x_1, x_2, y_1)$ 。对树 T 向下遍历的过程类似于在范围 $[x_1, x_2]$ 上进行一维范围查找。但是, 一个重要差别是: 当 $y(v) \geq y_1$ 时, 才会继续在结点 v 的子树中进行查找。算法12-4(PSTSearch)给出了三边范围查找算法的细节。图12-4说明了算法的执行过程。

算法12-4 优先查找树 T 中的三边范围查找。起初通过 $\text{PSTSearch}(x_1, x_2, y_1, T.\text{root}())$ 调用算法

算法 $\text{PSTSearch}(x_1, x_2, y_1, v)$:

输入: x_1, x_2 和 y_1 定义三边范围, 以及优先查找树 T 中的结点 v

输出: 存储在以 v 为根的子树中的数据项, 其坐标为 (x, y) , 满足 $x_1 \leq x \leq x_2$ 且 $y_1 \leq y$

if $\bar{y}(v) < y_1$ then

```

return  $\emptyset$ 
if  $x_1 \leq \bar{x}(v) \leq x_2$  then
     $M \leftarrow \{\bar{p}(v)\}$     {应该输出  $\bar{p}(v)$ }
else
     $M \leftarrow \emptyset$ 
if  $x_1 \leq \hat{x}(v)$  then
     $L \leftarrow \text{PSTSearch}(x_1, x_2, y_1, T.\text{leftChild}(v))$ 
else
     $L \leftarrow \emptyset$ 
if  $\hat{x}(v) \leq x_2$  then
     $R \leftarrow \text{PSTSearch}(x_1, x_2, y_1, T.\text{rightChild}(v))$ 
else
     $R \leftarrow \emptyset$ 
return  $L \cup M \cup R$ 

```

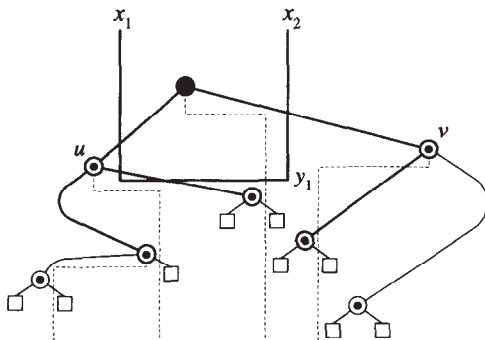


图12-4 优先查找树中的三边范围查找。访问的结点用粗线画出。存储输出数据项的结点则填充灰色

注意已经定义了三边范围是左边、右边和下边，上边无界。然而，对于利用矩形的任何三边定义的三边范围查询，所做的限制不失一般性。来自这种替代定义的优先查找树类似于上述的定义，但是会“打开它的边”。

我们分析方法PSTSearch的运行时间，回答存储二维关键字的 n 个数据项的优先查找树 T 上的三边范围查找查询问题。设 s 表示输出的数据项个数。因为访问每个结点的时间为 $O(1)$ ，方法PSTSearch的运行时间与访问的结点数的成正比。

558

可把方法PSTSearch访问的每个结点 v 分类如下：

- 把 T 看作 x 坐标的中值存储在其结点上的二叉查找树，如果结点 v 在 x_1 或 x_2 的查找路径上，则结点 v 是一个边界结点 (boundary node)。存储在内部边界结点中的数据项可能在三边之内，也可能在三边之外。由引理12.2可知， T 的高度为 $O(\log n)$ 。因此，有 $O(\log n)$ 个边界结点。
- 如果结点 v 是一个内部结点，而不是一个边界结点，且 $\bar{y}(v) \geq y_1$ ，则结点 v 是一个内结点 (inside node)。存储在内部结点中的数据项是在三边范围内。内结点数不超过输出数据项的个数 s 。
- 如果结点 v 不是边界结点，且如果它是内部结点，有 $\bar{y}(v) < y_1$ ，则结点 v 是一个终端结点 (terminal node)。存储在内部终端结点中的数据项是在三边范围外。每个终端结点是边界结点或内结点的一个子结点。因而，终端结点数至多是边界结点数与内结点数之和的两倍。因此，有 $O(\log n + s)$ 个终端结点。

由此可得PSTSearch访问 $O(\log n + s)$ 个结点, 这给出以下定理。

定理12.3 存储二维关键字的 n 个数据项的优先查找树 T 利用 $O(n)$ 的空间, 构建时间为 $O(n \log n)$, 利用 T , 三边范围查询所用的时间为 $O(\log n + s)$, 其中 s 是输出的数据项数。

当然, 三边范围查找不像规则的(四边)范围查询那样具有一般性, 规则范围查询利用上一节讨论的范围树的数据结构, 可用 $O(\log^2 n + k)$ 的时间解答问题。优先查找树仍可用于加快解答标准四边、二维范围查询的运行时间。所得到的数据结构称为优先范围树(priority tree), 利用优先查找树作为辅助结构, 在某种程度上达到传统范围树的同样的空间界限。以下讨论它的数据结构。

559

12.2.3 优先范围树

设 T 是存储二维关键字的 n 个数据项的平衡二叉查找树, 按照 x 坐标排序。我们显示如何用优先查找树作为辅助结构增大 T , 解答(四边)范围查询问题。所得到的数据结构称为优先范围树。

为了把 T 转换成优先范围树, 访问 T 中除根以外的每个结点, 并对于存储在 T 的以 v 为根的子树中的数据项, 构造一棵优先查找树 $T(v)$ 作为辅助结构, 如果 v 是一个左子结点, 则 $T(v)$ 解答右边无界的三边范围的范围查询。如果 v 是一个右子结点, 则 $T(v)$ 解答左边无界的三边范围的范围查询。由引理12.1和引理12.2可知, 优先范围树利用 $O(n \log n)$ 的空间, 构造时间为 $O(n \log n)$ 。算法12-5(PSTRangeSearch)中给出在优先范围树中进行二维范围查询的方法。

算法12-5 在优先范围树 T 中进行范围查找。起初通过PSTRangeSearch($x_1, x_2, y_1, y_2, T.root()$)调用算法

```

算法 PSTRangeSearch( $x_1, x_2, y_1, y_2, v$ ):
    输入: 查找关键字 $x_1, x_2, y_1$ 和 $y_2$ ; 优先范围树的主结构 $T$ 中的结点 $v$ 
    输出: 以 $v$ 为根的子树中的数据项, 其 $x$ 坐标和 $y$ 坐标分别在 $[x_1, x_2]$ 和 $[y_1, y_2]$ 中
    if  $T.isExternal(v)$  then
        return  $\emptyset$ 
    if  $x_1 \leq x(v) \leq x_2$  then
        if  $y_1 \leq y(v) \leq y_2$  then
             $M \leftarrow \{element(v)\}$ 
        else
             $M \leftarrow \emptyset$ 
         $L \leftarrow PSTSearch(x_1, y_1, y_2, T(leftChild(v)).root())$ 
         $R \leftarrow PSTSearch(x_2, y_1, y_2, T(rightChild(v)).root())$ 
        return  $L \cup M \cup R$ 
    else if  $x(v) < x_1$  then
        return PSTRangeSearch( $x_1, x_2, y_1, y_2, T.rightChild(v)$ )
    else
         $\{x_2 < x(v)\}$ 
        return PSTRangeSearch( $x_1, x_2, y_1, y_2, T.leftChild(v)$ )

```

定理12.4 存储二维关键字的 n 个数据项的优先范围树 T 利用 $O(n)$ 的空间, 构建时间为 $O(n \log n)$, 利用 T , 三边范围查找查询所用的时间为 $O(\log n + s)$, s 是输出的元素个数。

560

12.3 四叉树和 k -d 树

大型应用常常导致多维数据集; 因此, 我们想要利用线性空间结构存储它们。设计 d 维数

据的某种线性结构的一般框架基于称为划分树的方法, 假设维数 d 为固定的常数。

划分树 (partition tree) 是一棵有根树 T , 它至多有 n 个外部结点, 其中 n 是给定集合 S 中的 d 维点的个数。划分树 T 中的每个外部结点存储 S 中的不同小子集。划分树 T 中的每个内部结点 v 对应一个 d 维空间区域。然后这个 d 维区域被划分成关联 v 的子结点的不同单元或区域的某个数 c 。对于关联 v 的子结点 u 的每个区域 R , 要求在 u 的子树中的所有点落入区域 R 之内。理想情况下, 利用常数个比较和算术计算就能把 v 的子结点的 c 个不同单元区分开。

12.3.1 四叉树

我们讨论的第一个划分树的数据结构是四叉树 (quadtree)。四叉树的主要应用是来自图像中的点集, 因为数据点是图像像素, x 坐标和 y 坐标是整数。此外, 如果点的分布相当不均匀, 它们展示其最好性质, 某些区域大部分为空, 其他一些区域稠密。

给定平面上 n 个点的集合 S 。此外, 设 R 表示包含 S 中所有点的区域 (例如, R 可以是一个 2048×2048 的有界区域, 该区域产生了集合 S)。四叉树数据结构是一棵划分树 T , 满足 T 的根 r 关联区域 R 。为了得到 T 的下一层, 把 R 划分成四个相等大小的方形区域 R_1 、 R_2 、 R_3 和 R_4 , 每个方形区域 R_i 关联根 r 的一个可能的子结点。确切地讲, 如果方形区域 R_i 包含 S 中的一个点, 则创建 r 的一个子结点 v_i 。如果方形区域 R_i 没有包含 S 中的点, 则创建 r 的空子结点。把 R 逐步细化为方形区域 R_1 、 R_2 、 R_3 和 R_4 的过程称为分裂 (split)。

四叉树是通过递归地对 r 的每个子结点 v 进行分裂来定义的, 如果需要分裂的话。也就是说, r 的每个子结点有一个方形区域 R_i 关联它, 如果 v 的区域 R_i 包含 S 中的一个以上的点, 那么在 v 处进行分裂, 把 R_i 划分成四个相等大小的方形区域, 并重复上述在 v 处的划分过程。按照这种方式进行, 把包含多个点的方形区域分裂成四个子方形区域, 对非空子方形区域进行递归, 直到把 S 中的所有点分割成独立的方形区域。然后把 S 中的每个点存储在 T 的外部结点中。它对应划分过程中包含 p 的最小区域。在每个内部结点 v 上存储对 v 进行分裂的一个简洁表示。

561

我们说明点集的一个例子及图12-5中关联的四叉树。然而注意, 与说明相反, 四叉树的深度可能没有上界, 这与以前定义的一样。例如, 点集 S 可能包含彼此非常相近的两个点, 在把这两个点分开之前, 可能需要进行较长的分割序列。因此, 按惯例四叉树设计者会对树 T 的深度确定某个上界。给定平面上 n 个点的集合 S , 可以构造 S 的一棵四叉树 T , 用 $O(n)$ 的时间构造 T 的每一层。因此, 在最坏情况下, 构造这样一棵深度有界的四叉树需要 $O(Dn)$ 的时间。

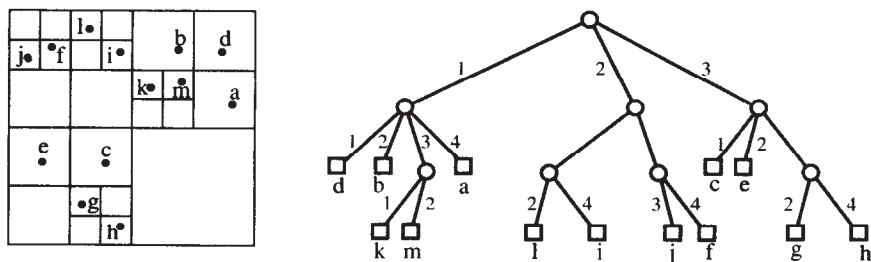


图12-5 四叉树。我们说明点集的一个例子及其对应的四叉树数据结构

1. 用四叉树解答范围查询

四叉树常常用于解答的查询之一是范围查找。给定矩形 A 按照坐标轴排列, 要求利用一棵四叉树 T 返回 S 中包含在 A 中的所有点。解答这个查询的方法相当简单。从树的根 r 开始, 比较 r 所在区域 R 与 A 。如果 A 和 R 根本不相交, 则完成查询——以 r 为根的子树中没有点落入 A 之内。另一方

面, 如果 A 完全包含 R , 那么简单地枚举出 r 的所有外部结点的后代。这些是两种简单的情况。如果 R 和 A 相交, 但 A 不完全包含 R , 那么对 r 的每个子结点 v 进行递归查找。

2. 性能

在进行范围查找查询中, 在最坏情况下, 可以遍历整棵树, 并且不产生任何输出。因此, 在深度为 D 且有 n 个外部结点的四叉树上进行范围查询最坏情况下的运行时间为 $O(Dn)$ 。从最坏情况下的观点来看, 用四叉树解答范围查找查询实际上比对集合进行蛮力查找还要糟糕, 它需要 $O(n)$ 时间解答一个二维范围查询。在实际中, 用四叉树进行范围查找查询的处理速度会更快。

562

12.3.2 k -d 树

四叉树有一个缺点, 这就是不能很好地推广到高维空间。尤其是, 类似四维空间的四叉树中的每个结点可能有16个子结点。 d 维四叉树中的每个结点可能有多达 2^d 个子结点。为了克服处理高于三维数据的存储问题, 数据结构设计者常常考虑另一种二叉树的划分树结构。

另一种划分数据结构是 k -d树, 与四叉树数据结构类似, 但它是二叉树。 k -d树数据结构实际上是一个划分树数据结构族, 所有树是存储多维数据的二叉划分树。像四叉树数据结构一样, k -d树中的每个结点 v 关联一个矩形区域 R , 尽管在 k -d树的情况下这个区域不必是方形的。差别在于对 k -d树中的结点 v 进行的分裂操作, 只用一条垂直于其中一个坐标轴的直线即可。对于三维或更高维的数据集合, 这条“线”是一条按轴排列的超平面。因此, 不论维数是多大, 一棵 k -d树总是一棵二叉树, 因为可以把 v 的区域 R 在分割线左边的部分关联 v 的左子结点, 以及把 v 的区域 R 在分割线右边的部分关联 v 的右子结点。如同四叉树结构中那样, 如果一个区域中的点数小于某个固定的常数阈值, 则停止分裂 (如图12-6所示)。

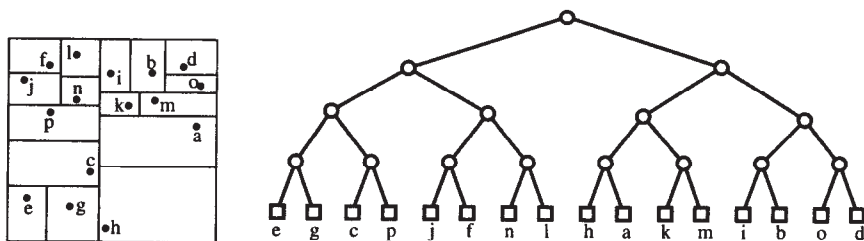


图12-6 k -d树示例

实质上有两种不同类型的 k -d树, 它们是基于区域 (region-based) 的 k -d树和基于点 (point-based) 的 k -d树。基于区域的 k -d树本质上是四叉树的一个二叉版本。在基于区域的 k -d树中, 每当需要分裂一个矩形区域时, 则用一条与 R 的最长边垂直的直线把区域 R 精确分成两半。如果 R 中的最长边多于一条, 则按照“轮流”方式分割它们。另一方面, 在基于点的 k -d树中, 基于矩形区域内点的分布进行分裂。图12-6所示的 k -d树是基于点的 k -d树。

563

在基于点的 k -d树中, 对包含子集 $S' \subseteq S$ 的矩形区域进行分裂的方法由两步组成。在第一步中, 根据 S' 中那些点在第 i 维上的最大差值, 来确定维数 i 。例如, 对于每个维数 j , 找出 S' 中具有最小和最大维数 j 值的点, 取 i 为这些值中两个值之间的最大差值。在第二步中, 从 S' 中的这些点确定中值维数 i , 用一条与第 i 维的轴垂直的直线穿过这个中值, 实现对 R 的分裂。因此, 对 R 的分裂把 S' 中的点集分为两半, 但可能对 R 分得不是很均匀。利用线性时间的找中值的方法 (4.7节) 进行这个分裂步骤所需的时间为 $O(k|S'|)$ 。于是, 构建 n 个点集的一棵 k -d树的运行时间可用以下递归方程表征: $T(n) = 2T(n/2) + kn$, 它是 $O(kn \log n)$ 。此外, 每次分裂都会把一个结点上关联的点集大小

分为两部分，因而树的高度为 $O(\log n)$ 。图12-6说明了利用这个算法构建的一棵基于点的 k -d树。

基于点的 k -d树的优点在于可以保证很好的深度和构造时间。这种模式的缺点是可能引起“长而细”的矩形区域，通常认为这种情况对于大多数 k -d树查询方法是不好的。实际中，这样长而细的区域很少出现，关联 k -d树结点的大多数矩形区域是“盒状的”。

利用 k -d树进行最近邻查找

我们讨论如何利用 k -d树解答查询问题。特别是，重点关注最近邻查找问题，给定一个查询点 p ，要求找出 S 中与 p 最近的点。利用 k -d树回答这样一个查询的良好方法如下。首先沿着树 T 向下查找，定位包含 p 的最小矩形区域的外部结点 v ，比较 S 中落入区域 R 或落入关联 v 的兄弟结点的区域中的那些结点，找出当前最近邻 q 。然后，定义一个以 p 为中心的球，包含 q 作为当前最近邻的球 s 。给定这个球，然后对 T 进行遍历（倾向于自底向上进行遍历），找出关联 T 的外部结点与 s 相交的区域。在遍历过程中，如果找到比 q 更近的点，那么把对 q 的引用更新到这个新点，并更新球 s ，使其包含这个新点。我们并不访问那些与 s 不相交的区域中的结点。当穷尽所有选择时，输出当前点 q ，作为 p 的最近邻的点。在最坏情况下，这个方法可能需要 $O(n)$ 的时间，但有许多不同的分析方法和实验分析表明，在对 S 中的点进行合理假设之下，它的平均运行时间更像 $O(\log n)$ 。此外，实际上有许多有用的启发式搜索方法可以加快这个查找过程，利用优先查找策略，可以以子树关联区域到 p 的距离的次序访问 T 的子树。

564

12.4 平面扫描技术

在这一节里，研究一种用于许多不同几何问题的技术。其主要思想是把一个静态的二维问题转变成一个动态的一维问题，利用插入、删除和查询操作序列求解问题。我们不是以抽象的方式呈现这种技术，而是用许多具体的例子说明它的用法。

12.4.1 正交线段相交

利用平面扫描技术解决的第一个问题是找出 n 条直线段的集合中的所有相交对。当然，可以利用蛮力算法检查每对线段，查看它们是否相交。因为有 $n(n-1)/2$ 对，而测试每一对线段是否相交的时间为常数，因此这个算法所需时间为 $O(n^2)$ 。如果所有对都相交，这个算法是最优的。然而，当相交线段的对数较少，或根本不存在相交的情况时，我们希望有更快的方法。确切地讲，如果 s 是相交对数，我们希望有一个输出敏感的算法，它与 n 和 s 都有关。利用平面扫描技术，我们提出一种算法的运行时间为 $O(n \log n + s)$ ，它是针对线段的输入集合由 n 条正交线段（orthogonal segment）组成的情况。这意味着集合中的每条线段要么水平要么垂直。

1. 改进的一维范围查找

在处理算法之前，稍微偏离一点主题，回顾本章前面讨论的一个问题。

这个问题是一维范围查找问题，在这个问题中，我们希望动态维持一个数的字典（即在一条数值线上的点），其插入、删除和查询具有如下形式：

- $\text{findAllInRange}(k_1, k_2)$ ：返回 D 中满足 $k_1 \leq k \leq k_2$ 的关键字 k 对应的所有元素。

12.1.1节显示了如何利用平衡二叉查找树（如AVL树或红黑树）维持这样一个字典，使点的插入和删除时间为 $O(\log n)$ ，解答 findAllInRange 查询的时间为 $O(\log n + s)$ ，其中 n 是此时字典中的点数， s 是返回的那个范围中的点数。我们将不会利用算法的细节，只认为它存在。因此跳过12.1.1节的读者可以确信这个结果，而不必关心它是如何达到这个结果的。

565

2. 范围查找问题集

我们回到所讨论的问题，从 n 条水平线和垂直线计算所有相交的线段对，求解这个问题的算法的主要思想是把这个二维问题归约为一维范围查找问题集，即对于每条垂直线段 v ，考虑通过 v 的垂线 $l(v)$ ，并投入到直线 $l(v)$ 的“一维世界”（见图12-7a和b。）在这个世界中，只有垂直线段 v 以及与 $l(v)$ 相交的水平线段。特别是，线段 v 对应 $l(v)$ 的一个区间。与 $l(v)$ 相交的水平线段 h 对应 $l(v)$ 上的一个点，穿过 v 的水平线段对应 $l(v)$ 上所包含的那个区间上的点。

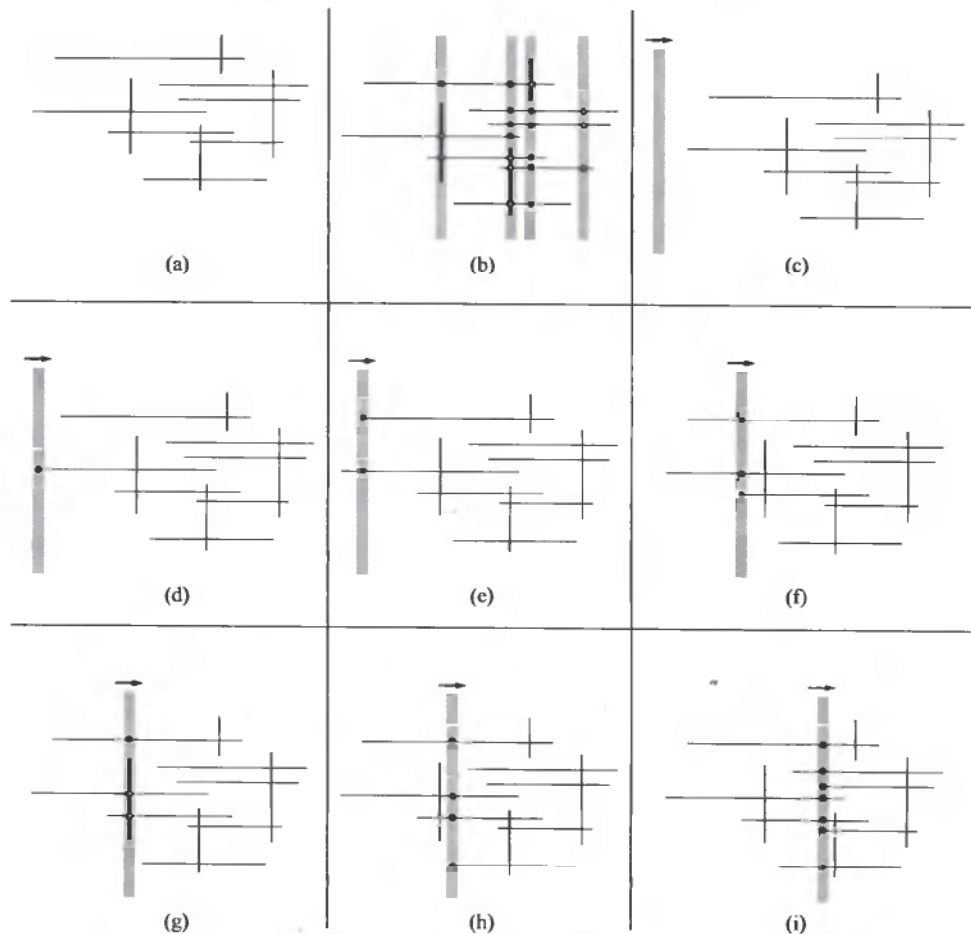


图12-7 正交线段相交的平面扫描算法：(a)水平线段和垂直线段集合；(b)一维范围查找问题集；(c)开始平面扫描（水平线段的有序字典 S 为空）；(d)第一个（左端点）事件，引起向 S 中的插入；(e)第二个（左端点）事件，引起向 S 中的另一次插入；(f)第三个（左端点）事件，又引起向 S 中的插入行为；(g)第一个垂直线段事件，引起在 S 中进行范围查找（输出两条线段的交）；(h)下一个（左端点）事件，引起向 S 中的插入；(i)三个事件之后的左端点事件，引起向 S 中的插入

因此，如果给定与直线 $l(v)$ 相交的水平线段的集合 $S(v)$ ，那么确定与线段 v 相交的那些线段等价于利用线段的 y 坐标作为关键字，并线段 v 的 y 坐标的端点给出的区间作为选择范围，在 $S(v)$ 中进行范围查找。

3. 平面扫描线段相交算法

给定平面上 n 条水平线段和垂直线段的集合。我们要利用平面扫描技术和上述思想提出的集

合方法, 确定这个集合中所有相交线段对。这个算法涉及模拟一条垂线 l 的扫描过程, 从所有输入线段左边的一个位置开始, 在线段上从左向右移动。在扫描过程中, 通过在按照 y 坐标排序的字典中执行插入和删除, 维持当前与扫描线相交的水平线段集。当扫描线遇见一条垂直线段 v 时, 就对字典进行范围查询, 找与 v 相交的水平线段。

确切地讲, 在扫描过程中维持一个存储水平线段及其按 y 坐标给出关键字的有序字典 S 。扫描过程在某个事件(event)处停止, 该事件引发表12-1显示的行为(action)。图12-7说明了这些行为。

表12-1 在用于正交线段相交的平面扫描算法中, 事件引发的行为

事 件	行 为
水平线段 h 的左端点	把 h 插入字典 S 中
水平线段 h 的右端点	从字典 S 中删除 h
垂直线段 v	对 S 进行范围查找, v 的端点的 y 坐标给出了选择范围

566
567

4. 性能

为了分析这个平面扫描算法的运行时间, 首先注意必须确定所有事件, 并按照 x 坐标对它们排序。一个事件要么是水平线段的一个端点, 要么是垂直线段的一个端点。因此, 事件数至多为 $2n$ 。当对事件排序时, 按照 x 坐标对它们进行比较, 这需要 $O(1)$ 的时间。利用一个渐近最优的排序算法, 如堆排序(2.4.4节)或归并排序(4.1节), 可在 $O(n \log n)$ 时间内完成对事件的排序。对字典 S 进行的操作有插入、删除和范围查找。每次执行一个操作时, S 的大小至多为 $2n$ 。我们用一棵AVL树(3.2节)或者红黑树(3.3.3节)实现 S , 每次进行的插入和删除操作所需时间为 $O(\log n)$ 。正如上述的回顾(见12.1.1节), 在一个具有 n 个元素的有序字典中进行范围查找所需时间为 $O(\log n + s)$, 利用 $O(n)$ 的空间, 其中 s 是输出的数据项数。然后我们表征垂直线段 v 引发的范围查找的运行时间为 $O(\log n + s(v))$, 其中 $s(v)$ 是当前字典 S 中与 v 相交的水平线段数。因此, 设 V 表示垂直线段集合, 扫描的运行时间为

$$O\left(2n \log n + \sum_{v \in V} (\log n + s(v))\right)$$

因为扫描会经过所有线段, 在遇见的所有垂直线段上, $s(v)$ 的和等于相交线段对的总数 s 。因此, 可得扫描时间为 $O(n \log n + s)$ 。

总之, 上面概括的完整线段相交算法由事件排序步骤接着扫描步骤组成。对事件排序所需时间为 $O(n \log n)$, 而扫描时间为 $O(n \log n + s)$ 。因此, 算法的运行时间为 $O(n \log n + s)$ 。

12.4.2 查找最近点对

利用平面扫描技术解决的另一个几何问题涉及接近程度(proximity)的概念, 它是一种存在于几何对象之间的距离(distance)关系。确切地讲, 我们集中关注最近点对(closest pair)问题, 它是在 n 个点的集合中, 找出具有最小距离的点对 p 和 q 。称这个点对为最近点对。我们利用欧氏距离定义两个点 a 和 b 之间的距离:

$$\text{dist}(a, b) = \sqrt{(x(a) - x(b))^2 + (y(a) - y(b))^2}$$

其中 $x(p)$ 和 $y(p)$ 分别表示点 p 的 x 坐标和 y 坐标。最近点对问题的应用包括机械部件的验证和集成电路的验证, 在这些应用中考虑部件之间的某种分离规则是重要的。

568

平面扫描算法

解决最近点对问题的直接“蛮力”算法是计算每一对点之间的距离，并选择具有最小距离的一对点。因为有 $n(n-1)/2$ 对，算法所需时间为 $O(n^2)$ 。然而我们可以利用更聪明的策略，避免检查所有的点对。

我们可以把平面扫描技术有效地应用于最近点对问题。在这种情况下，解决最近点对问题的方式是：想象用一条垂线从所有 n 个输入点的左边位置开始，从左向右扫描平面。当扫描线穿过平面时，记录到目前为止看到的最近点对，以及扫描线“附近”的所有那些点。我们还记录到目前为止看到的最近点对之间的距离 d 。尤其是，如图12-8中所说明的，当扫描从左向右穿过点时，维持以下数据：

- 所遇见点之间的最近点对 (a, b) ，以及距离 $d = \text{dist}(a, b)$
- 有序字典 S ，存储位于扫描线左边宽为 d 的条带中的点，并利用点的 y 坐标作为关键字。

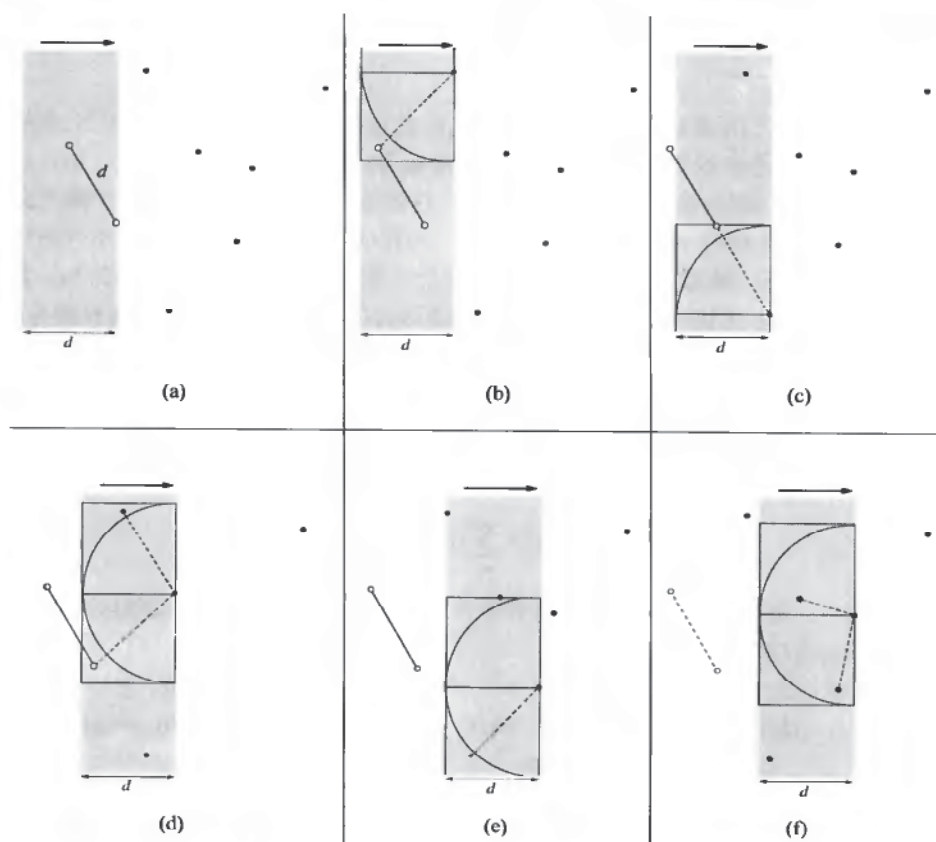


图12-8 最近点对问题的平面扫描：(a)第一个最短距离 d 和最近点对（突出显示）；(b)下一个事件（矩形 $B(p, d)$ 包含一个点，但半圆 $C(p, d)$ 中为空）；(c)下一个事件（ $C(p, d)$ 再次为空，但从 S 中删除一个点）；(d)下一个事件（ $C(p, d)$ 再次为空），字典 S 包含的点位于宽度为 d 的灰色条带中；(e)遇见点 p （且从 S 中删除一个点）， $B(p, d)$ 中包含一个点，但 $C(p, d)$ 中不含点；因此，最小距离 d 和最近点对 (a, b) 不变；(f)遇见点 p ， $C(p, d)$ 中包含两个点（且从 S 中删除一个点）

每个输入点 p 对应这个平面扫描中的一个事件。当扫描线遇见一个点 p 时，进行如下行为：

- (1) 删除与 p 的水平距离大于 d 的那些点，更新字典 S ，即删除那些 $x(p) - x(r) > d$ 的每个点 r 。

(2) 查找字典 S , 找出距离 p 的左边最近的点 q (稍候介绍如何做到)。如果 $\text{dist}(p, q) < d$, 那么设置 $a \leftarrow p$, $b \leftarrow q$, 且 $d \leftarrow \text{dist}(p, q)$, 更新当前最近点对以及距离。

(3) 把 p 插入 S 中。

显然可以把对距离 p 的左边最近点 q 的查找限制于字典 S 中的点, 因为所有其他点的距离都会大于 d 。我们希望的是那些 S 中位于半径 d 的半圆 $C(p, d)$ 中的点, 该半圆以 p 点为圆心, 位于 p 的左边 (如图12-9所示)。作为第一次近似, 在 S 上进行范围查找 (12.1.1节), 查找 S 中那些其 y 坐标在关键字区间 $[y(p)-d, y(p)+d]$ 中的点, 得到 $C(p, d)$ 的 $d \times 2d$ 封闭矩形区域 $B(p, d)$ 中的点 (如图12-9所示)。我们逐个检查这些点, 找到距离 p 最近的点, 用 q 表示。因为在字典 S 上进行的操作是范围查找、插入点和删除点, 因而可以用AVL树或红黑树实现 S 。

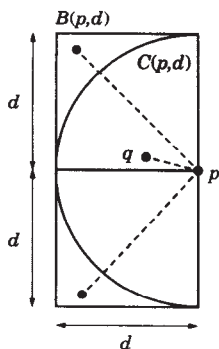


图12-9 矩形 $B(p, d)$ 和半圆 $C(p, d)$

以下直观性质的证明留作习题 (R-12.3), 它对于分析算法的运行时间非常关键。

定理12.5 宽为 d 、高为 $2d$ 的矩形区域至多包含6个点, 满足任意两个点的距离至少为 d 。

因此, S 中至多有6个点位于矩形 $B(p, d)$ 中。因此对 S 进行范围查找, 找出 $B(p, d)$ 中的点所需时间为 $O(\log n + 6)$, 它是 $O(\log n)$ 。此外, 找出 $B(p, d)$ 中与 p 最近的点所需时间为 $O(1)$ 。

在开始扫描之前, 按照点的 x 坐标对点排序, 并把它们存储在有序表 X 中。使用有序表 X 出于两个目的:

- 取下一个要处理的点
- 确定要从 S 中删除的点

我们保持对表 X 中两个位置的引用。分别用 firstInStrip 和 lastInStrip 表示。位置 lastInStrip 记录要被插入到 S 中的新点, 而位置 firstInStrip 记录 S 中最左边的点。通过一次一步推进 lastInStrip , 可以找到要被处理的新点。利用 firstInStrip , 可以找出从 S 中要出删除的点, 即当

$$x(\text{point}(\text{firstInStrip})) < x(\text{point}(\text{lastInStrip})) - d$$

时, 在字典 S 上执行操作 $\text{removeElement}(y(\text{point}(\text{firstInStrip})))$, 并推进 firstInStrip 。

设 n 是输入点数。对最近点对的平面扫描算法的分析基于以下观察:

- 按 x 坐标排序所需时间为 $O(n \log n)$ 。
- 每个点都会向 S 中插入一次, 并从 S 中删除一次。 S 大小至多为 n ; 因此, S 中插入元素和删除元素的总时间为 $O(n \log n)$ 。
- 由定理12.5可知, 在 S 中每次进行范围查询的时间为 $O(\log n)$ 。每处理一个新点, 执行一次这样的范围查询。因此, 进行范围查询花费的总时间为 $O(n \log n)$ 。

由此可得计算 n 个点的集合中的最近邻点对的时间为 $O(n \log n)$ 。

12.5 凸包

最常研究的几何问题之一是计算点集的一个凸包。非形式地讲，平面点集的凸包是用一个橡皮圈“沿着点”放置，并允许它收缩到平衡状态产生的形状（见图12-10）。

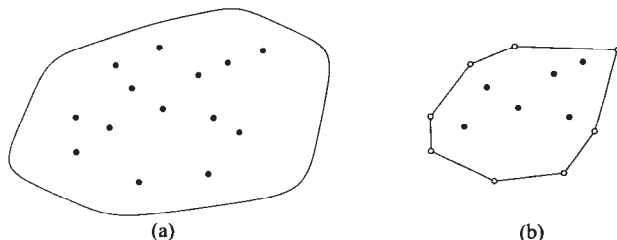


图12-10 平面点集的凸包：(a)“橡皮圈”沿着点放置；(b)点的凸包

凸包对应点集“边界”的直观概念，可用于近似复杂对象的外形。计算点集的凸包是计算几何中的基本操作。在详细描述凸包和计算它的算法之前，首先需要讨论几何数据对象的一些表示问题。

12.5.1 几何对象表示

几何算法取各种类型的几何对象作为输入。平面上的基本几何对象有点、直线、线段和多边形。

有许多表示平面几何对象的方式。我们不是像在关于几何算法的书籍中那样给出点、直线、线段和多边形各自的ADT，而是假设已有这些对象的直观表示。即便如此，我们还是简略地介绍关于几何表示所做的一些选择。

可用坐标对 (x, y) 表示平面上的一个点，存储那个点的 x 和 y 的笛卡儿坐标。虽然这种表示相当通用，但不是唯一表示。可能还有一些应用，用另一种表示方法更好（如把一个点表示为两条不平行直线的交点）。

572

直线、线段和多边形

可把一条直线 l 表示为一个三元组 (a, b, c) ，满足这些值是关联 l 的直线方程的系数。

$$ax + by + c = 0$$

另一方面，可以确定两个不同点 q_1 和 q_2 以及关联它们的直线，这条直线通过这两个点。给定 q_1 的笛卡儿坐标 (x_1, y_1) 和 q_2 的笛卡儿坐标 (x_2, y_2) 。通过 q_1 和 q_2 的直线 l 的方程为

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1}$$

由此可导出

$$a = (y_2 - y_1); \quad b = -(x_2 - x_1); \quad c = y_1(x_2 - x_1) - x_1(y_2 - y_1)$$

可用构成 s 端点的点对 (p, q) 表示平面上的线段 s 。也可用通过它的直线以及 x 坐标和 y 坐标的范围表示 s ，这个范围把直线限制为线段 s （仅仅只包括 x 坐标和 y 坐标的范围是不够的，为什么？）。

可用点的环形序列表示一个多边形 P ，称它们为 P 的顶点（vertex）（见图12-11）。称 P 的两个连续顶点之间的线段为 P 的边（edge）。如果 P 的一对边仅在公共端点处相交，则称多边形 P 是不相交的（nonintersecting），或是简单的（simple）。如果多边形是简单的且它的所有内角都小于 π ，则它是凸（convex）多边形。

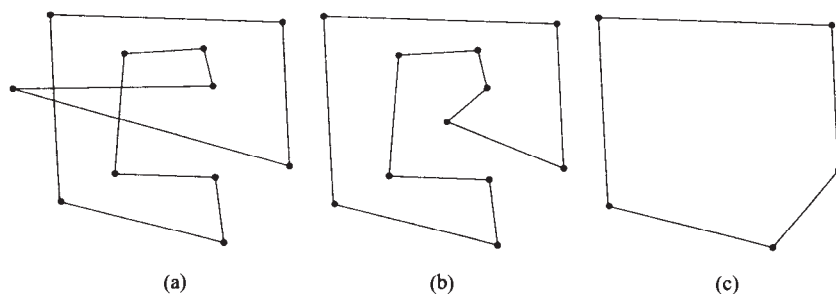


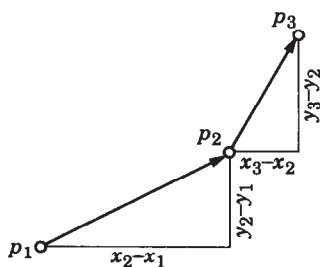
图12-11 多边形的例子: (a)相交多边形; (b)简单多边形; (c)凸多边形

我们并未彻底无遗漏地讨论点、线、线段和多边形的不同表示,而只是简单地表明实现这些几何对象的不同方法。

573

12.5.2 点方位测试

在许多几何算法(尤其是凸包构造)中出现的一种重要的几何关系是方位(orientation)。给定有序点的三元组 (p, q, r) ,如果从 p 到达 q 再到达 r 时的左边的角度小于 π ,则称 (p, q, r) 沿着逆时针方向(counterclockwise)进行一次左转(left turn)。如果右边的角度小于 π ,则称 (p, q, r) 沿着顺时针方向(clockwise)进行一次右转(right turn)(见图12-12)。可能左边的角度和右边的角度都等于 π 。在这种情况下,这三点实际上并没有转动,则称它们的方位是共线的(collinear)。

图12-12 左转的一个例子。同时说明了 p_1 和 p_2 之间以及 p_2 和 p_3 之间的坐标差值

给定平面上三个点 $p_1 = (x_1, y_1)$ 、 $p_2 = (x_2, y_2)$ 和 $p_3 = (x_3, y_3)$ 的三元组 (p_1, p_2, p_3) 。设 $\Delta(p_1, p_2, p_3)$ 由下式

$$\Delta(p_1, p_2, p_3) = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 - x_2y_1 + x_3y_1 - x_1y_3 + x_2y_3 - x_3y_2 \quad (12.1)$$

确定。常称函数 $\Delta(p_1, p_2, p_3)$ 为“有符号的面积”函数,因为它的绝对值是点 p_1 、 p_2 和 p_3 所形成(可能退化)的三角形面积的两倍。此外,可得以下此函数与方位测试的关系的重要事实。

定理12.6 平面上点的三元组 (p_1, p_2, p_3) 的方位是逆时针的、顺时针的或共线的,这分别取决于 $\Delta(p_1, p_2, p_3)$ 是为正、为负,还是为零。

我们概述定理12.6的证明;证明细节留作习题(R-12.4)。在图12-12中,显示一个满足 $x_1 < x_2 < x_3$ 的点的三元组 (p_1, p_2, p_3) 。显然,如果线段 p_2p_3 的斜率大于线段 p_1p_2 的斜率,那么这个三元组进行左转。这可表达为以下方程:

$$\frac{y_3 - y_2}{x_3 - x_2} > \frac{y_2 - y_1}{x_2 - x_1} \text{ 吗?} \quad (12.2)$$

[574] 展开方程12.1中显示的 $\Delta(p_1, p_2, p_3)$, 可以验证不等式12.2等价于 $\Delta(p_1, p_2, p_3) > 0$ 。

示例12.1 利用方位表示, 考虑测试两条线段 s_1 和 s_2 是否相交的问题。确切地讲, 设 $s_1 = \overline{p_1 q_1}$ 和 $s_2 = \overline{p_2 q_2}$ 是平面上的两条线段。当且仅当以下两个条件之一得到验证时, s_1 和 s_2 才相交:

- (1) (a) (p_1, q_1, p_2) 和 (p_1, q_1, q_2) 方位不同, 且
 (b) (p_2, q_2, p_1) 和 (p_2, q_2, q_1) 方位不同。
- (2) (a) (p_1, q_1, p_2) 、 (p_1, q_1, q_2) 、 (p_2, q_2, p_1) 和 (p_2, q_2, q_1) 都共线, 且
 (b) s_1 和 s_2 在 x 方向的投影相交, 且
 (c) s_1 和 s_2 在 y 方向的投影相交,

图12-13说明了条件(1)。同样在表12-2中对于条件(1)中的四种情况, 显示了三元组 (p_1, q_1, p_2) 、 (p_1, q_1, q_2) 、 (p_2, q_2, p_1) 和 (p_2, q_2, q_1) 各自的方位。完整的证明留作习题 (R-12.5)。注意如果 s_1 和/或 s_2 是一条具有重合端点的退化线段, 那么条件还是成立。

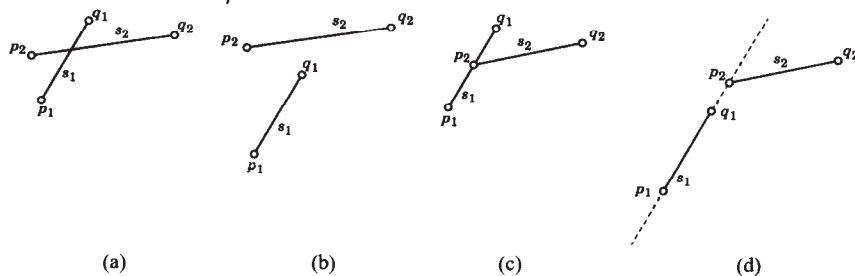


图12-13 说明示例12.1中条件(1)的四种情况的例子

表12-2 对于示例12.1中条件1确定的方位, 图12-13所示的四种情况。CCW代表逆时针, CW代表顺时针, COLL代表共线

情况	(p_1, q_1, p_2)	(p_1, q_1, q_2)	(p_2, q_2, p_1)	(p_2, q_2, q_1)	相交?
(a)	CCW	CW	CW	CCW	是
(b)	CCW	CW	CW	CW	否
(c)	COLL	CW	CW	CCW	是
(d)	COLL	CW	CW	CW	否

[575]

12.5.3 凸包的基本性质

如果对于区域 R 中的任意两个点 p 和 q , 整条线段 \overline{pq} 也在 R 中, 则称 R 是凸 (convex) 区域。点集 S 的凸包 (convex hull) 是包含 S 内及其边界上所有点的最小凸区域的边界。“最小”概念指的是周长或区域面积。这两个定义是等价的。平面点集 S 的凸包定义了一个凸多边形, 凸包边界上 S 中的点定义了这个凸多边形的顶点。以下例子描述了凸包在机器人运动规划问题中的应用。

示例12.2 机器人中的普遍问题是从一个起始点 s 到一个目标点 t , 确定一条避免障碍的轨道。在多条可能的轨道中, 想要找出一条尽可能短的轨道。假设障碍是一个多边形 P 。可以利用如下策略 (见图12-14), 计算出避免 P 的从 s 到 t 的最短轨道:

- 确定线段 $l = \overline{st}$ 是否与 P 相交。如果不相交,那么 l 是避开 P 的最短轨道。
- 否则,如果 $l = \overline{st}$ 与 P 相交,那么计算多边形 P 的顶点的凸包 H ,以及 s 和 t 。注意 s 和 t 把凸包 H 划分成两条多边形链,一条沿顺时针方向从 s 到 t ,另一条沿逆时针方向从 s 到 t 。
- 选择并返回 H 上这两条多边形链中最短的一条,且端点为 s 和 t 。

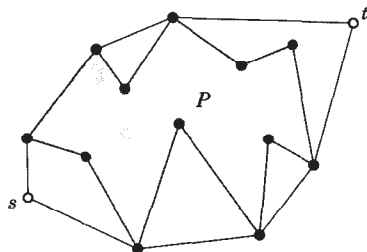


图12-14 从点 s 到 t 的避开障碍 P 的最短轨迹例子;轨迹是从 s 到 t 的顺时针链

最短的链就是平面上的避开障碍 P 的最短路径。

576

有许多凸包问题的应用,包括划分问题、形状测试问题和分割问题。例如,如果想要确定是否存在一个半平面(即在一侧的区域),完全包含点集 A ,但完全避免了点集 B ,只需计算 A 和 B 的凸包,并确定它们是否相交就足够了。

凸包有许多有趣的几何性质。以下定理提供了对点是否在凸包上的另一种表征。

定理12.7 设 S 是凸包为 H 的平面点集。那么

- 当且仅当 S 中的所有其他点都被包含在经过 a 和 b 的直线的一边时, S 中的点对 a 和 b 构成 H 的一条边。
- 当且仅当存在通过 p 的直线 l ,使得 S 中的所有其他点都包含在 l 分割的同一半平面上时(即它们都在 l 的同一边), S 中的点 p 是 H 的一个顶点。
- 当且仅当 p 被包含在 S 的其他三个点形成的三角形的内部,或者在 S 的其他两个点形成线段的内部, S 中的一个点 p 不是 H 的一个顶点。

定理12.7表达的性质在图12-15中做了说明。对它们的完整证明留作习题(R-12.6)。作为定理12.7的结果,可以直接验证,对于平面上的任意点集,以下关键(critical)点总是在 S 的凸包的边界上:

- 具有最小 x 坐标的点。
- 具有最大 x 坐标的点。
- 具有最小 y 坐标的点。
- 具有最大 y 坐标的点。

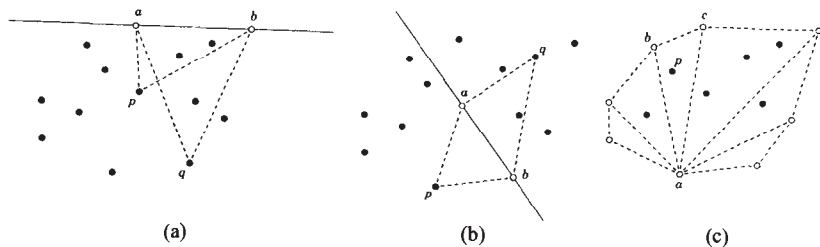


图12-15 定理12.7中给出的凸包性质的说明:(a)点 a 和 b 构成凸包的一条边;(b)点 a 和 b 不构成凸包的一条边;(c)点 p 不在凸包上

577

12.5.4 礼品包扎算法

定理12.7主要阐述了我们可以确定一个特殊点，如具有最小 y 坐标的点，为计算凸包的算法提供了一种初始起始配置。计算平面点集的凸包的礼品包扎（gift wrapping）算法基于这样一个起始点，直观描述如下（见图12-16）：

- (1) 把点看作插入底层的柱子，想象在柱子上系一根绳子，对应具有最小 y 坐标的点 a （和具有最小 x 坐标的点，如果存在tie）。称 a 为锚点（anchor point），注意 a 是凸包上的一个顶点。
- (2) 沿锚点右侧拉动绳子，并沿逆时针方向旋转，直至碰到另一根柱子，对应凸包上的下一个顶点。
- (3) 继续沿逆时针方向旋转绳子，每一步确定凸包上的一个新顶点，直到绳子返回到锚点。

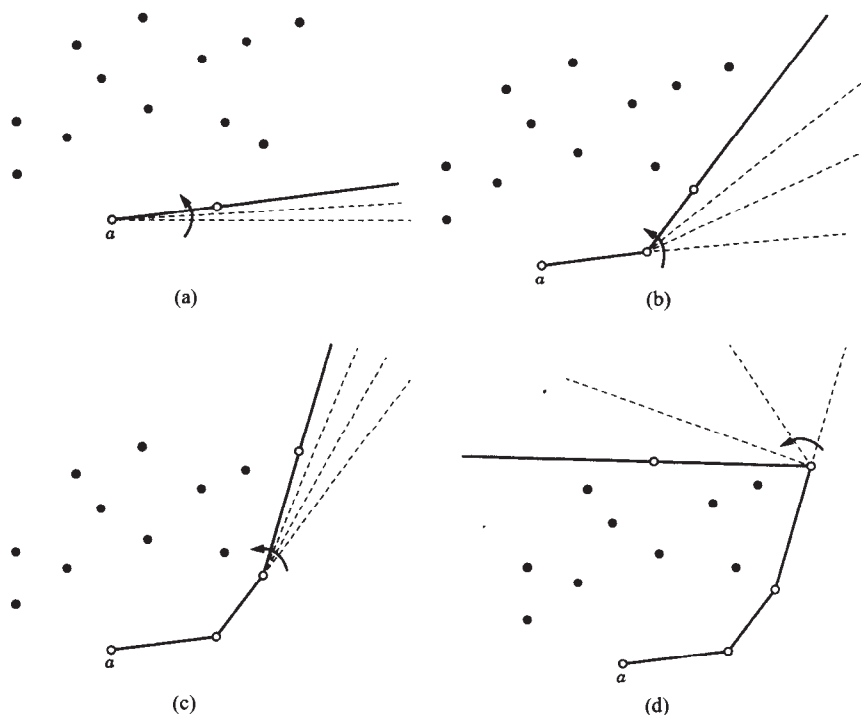


图12-16 礼品包扎算法的初始四个包扎步骤

578

每次沿着当前柱子旋转“绳子”时，会碰到另一个点，进行称为包扎步骤的操作。从几何上讲，从与凸包上当前锚点 a 相切的给定直线 L 开始包扎步骤，确定经过 a 点和集合中另一个点的直线，使得该直线与 L 成最小角度。实现这个包扎步骤并不需要三角几何函数和角度计算。而是利用以下定理进行包扎步骤，此定理可由定理12.11推出。

定理12.8 设 S 是平面上的点集， a 是 S 中的一点，它是 S 的凸包 H 上的顶点。从 a 开始沿着它的逆时针方向到达 H 的下一个顶点 p ，满足三元组 (a, p, q) 关于 S 中的每个其他顶点进行左转。

回忆2.4.1节的讨论，定义比较器 $C(a)$ ，利用 (a, p, q) 的方位比较 S 中的两个点 p 和 q 。也就是说，如果三元组进行左转，则 $C(a).isLess(p, q)$ 返回真。称比较器 $C(a)$ 为径向（radial）比较器，因为它根据锚点 a 周围的径向关系对点进行比较。由定理12.8可知，凸包上沿着点 a 的逆时针方向之后的顶点是关于径向比较器 $C(a)$ 的最小点。

性能

现在分析礼品包扎算法的运行时间。设 n 是 S 中的点数，并设 $h \leq n$ 是 S 的凸包 H 中的顶点数。设 p_0, \dots, p_{h-1} 是 H 中的顶点。找出锚点 $a = p_0$ 的时间为 $O(n)$ 。因为对于算法的每一个包扎步骤，都会发现凸包上的一个新顶点，包扎步骤数等于 h 。第 i 步是基于径向比较器 $C(p_{i-1})$ 找出最小值的计算，运行时间为 $O(n)$ ，因为确定三元组的方位需要 $O(1)$ 的时间，且找出关于 $C(p_{i-1})$ 的最小值这一步必须检查 S 中的所有点。由此可得，礼品包扎算法的运行时间为 $O(hn)$ ，最坏情况下它是 $O(n^2)$ 。当 $h = n$ 时，礼品包扎算法出现最坏情况，即所有点都在凸包上。

按照 n 的度量，礼品包扎算法最坏情况下的运行时间不是非常有效。然而这个算法在实际中还是合理有效的，因为它利用了当凸包点数 h 相对输入数较小时的（普遍）情况。也就是说，这个算法是一个输出敏感（output sensitive）的算法——它是一种运行时间取决于输出大小的算法。礼品包扎算法的运行时间在线性时间和二次时间之间，如果凸包上没有几个顶点，它就是有效算法。在下一节里，我们看到一个对所有凸包大小都会有效运行的算法，尽管它稍微有些复杂。

579

12.5.5 Graham 扫描算法

Graham扫描（Graham scan）算法是一种凸包算法，不论凸包边界上有多少顶点，它都有有效的运行时间。计算平面上 n 个点的集合 P 的凸包 H 的Graham扫描算法由以下三个阶段组成：

(1) 找出 P 中的点 a ，它是 H 的一个顶点，称之为锚点（anchor point）。例如，可以取 P 中具有最小 y 坐标的点作为锚点 a （以及具有最小 x 坐标的点，如果存在tie）。

(2) 利用径向比较器 $C(a)$ 对 P 中其余点（即 $P - \{a\}$ ）进行排序，并设 S 是对点排序的结果序列（见图12-17）。在表 S 中， P 中的点按照关于锚点 a 的“角度”逆时针排序，但比较器不做真实的角度计算。

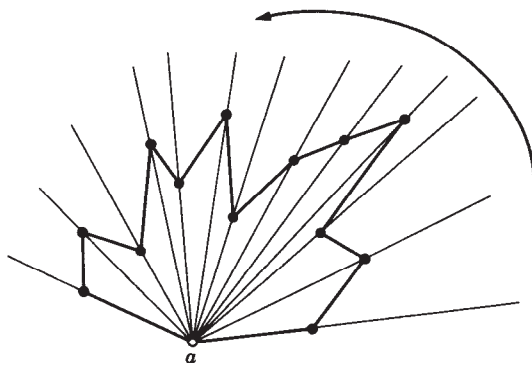


图12-17 在Graham扫描算法中沿着锚点排序

(3) 把锚点 a 添加到 S 中的第一个位置和最后一个位置，按照径向次序扫描 S 中的点，在每一步维持表 H ，它存储到目前为止“围绕”点扫描的凸包链。每当考虑一个新点 p 时，进行以下测试：

- (a) 如果 p 形成关于 H 中最后两个点的一个左转，或者如果 H 包含少于两个点，那么把 p 添加到 H 的末尾。
- (b) 否则，删除 H 中最后一个点，重复对 p 进行的测试。

当回到锚点 a 时，算法停止。此时 H 中按照逆时针方向存储 P 的凸包中的顶点。

算法12-6中描述的算法Scan清楚说明了扫描阶段（第三阶段）的细节（见图12-18）。

580

算法12-6 Graham凸包扫描算法的扫描阶段（见图12-18）。变量 $prev$ 、 $curr$ 和 $next$ 是表 S 中的位置（2.2.2节）。假设定义祖先方法 $point(pos)$ ，返回存储在位置 pos 处的点。算法的简化描述仅适合于 S 中至少有三个点且不存在三点共线的情况

算法 $Scan(S, a)$:

输入：平面上点的表 S ，起始点为 a ，满足 a 在 S 的凸包上。 S 中的其余点围绕 a 按照逆时针排序

输出：只含凸包顶点的表 S

$S.insertLast(a)$ {把 a 的副本添加到 S 的末尾}

$prev \leftarrow S.first()$ {使得起初 $prev = a$ }

$curr \leftarrow S.after(prev)$ {当前凸包链中的下一个点}

repeat

$next \leftarrow S.after(curr)$ {推进}

if 点($point(prev)$, $point(curr)$, $point(next)$)形成左转 **then**

$prev \leftarrow curr$

else

$S.remove(curr)$ {点 $curr$ 不在凸包中}

$prev \leftarrow S.before(prev)$

$curr \leftarrow S.after(prev)$

until $curr = S.last()$

$S.remove(S.last())$ {删除 a 的副本}

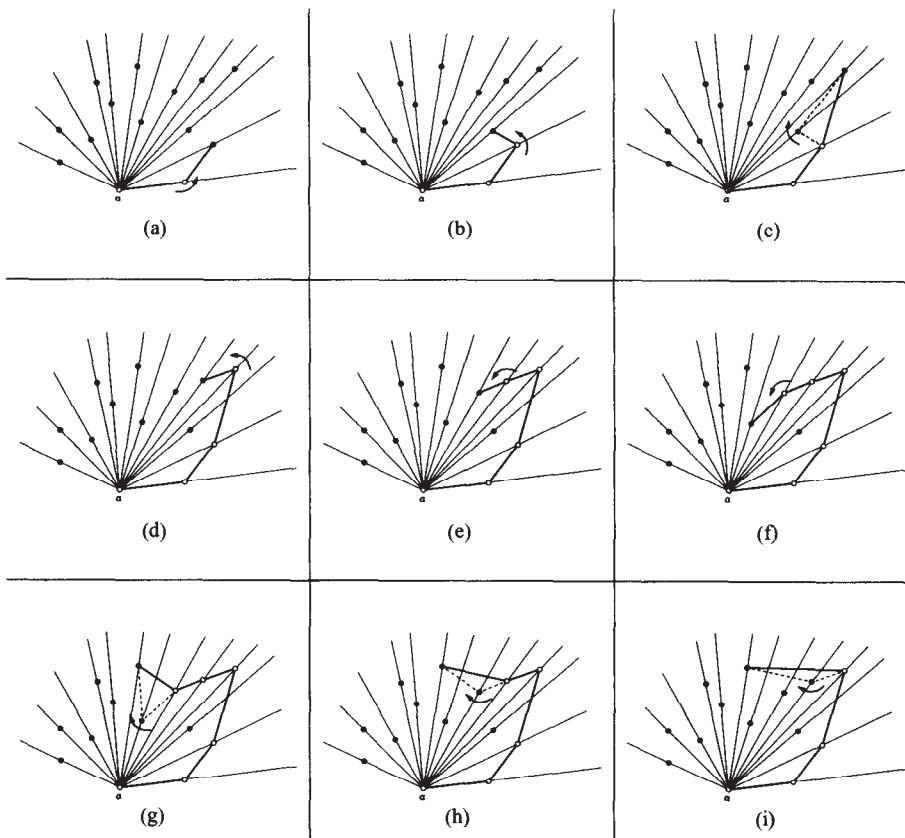


图12-18 Graham扫描算法的第三阶段（见算法12-6）

性能

现在分析Graham扫描算法的运行时间。设 n 表示 P (和 S)中的点数。显然第一阶段(找出锚点)所需时间为 $O(n)$ 。第二阶段(沿着锚点对点排序)如果利用渐近最优的排序算法,如堆排序(2.4.4节)或归并排序(4.1节),则所需时间为 $O(n \log n)$ 。对扫描阶段(第三阶段)的分析更细致。

为了分析Graham扫描算法的扫描阶段,我们更仔细地观察算法12-6中的repeat循环。在循环的每次迭代中,变量 $next$ 要么在表 S 中前进一个位置(如果测试成功),要么保持在原位,但从 S 中删除一个点(如果测试不成功)。因此,repeat循环的迭代次数至多为 $2n$ 。于是,算法Scan中的每条语句至多执行 $2n$ 次。因为每条语句的执行时间为 $O(1)$ 个基本操作,算法Scan所需时间为 $O(n)$ 。总之,Graham扫描算法的运行时间由第二阶段支配,这一阶段执行排序过程。因此,Graham扫描算法的运行时间为 $O(n \log n)$ 。

581
582

12.6 Java 示例:凸包

我们描述了Graham扫描算法(算法12-6)。假设没有退化,即排除输入配置的特殊情况(如点重合或共线)。然而,在实现Graham扫描算法时,重要的是要处理所有可能的输入配置。例如,两个或更多点是重合的,以及某三个点可能共线。

在代码段12-1~12-3中,显示了Graham扫描算法的Java实现。主方法是`grahamScan`(代码段12-1),利用了几个辅助方法。由于可能出现退化点配置,方法`grahamScan`处理了几种特殊情况。

- 首先把输入表复制到序列`hull`中,它在执行结束时返回(代码段12-2中的方法`copyInputPoints`)。
- 如果输入为零个或一个点,则返回(输出与输入相同)。
- 如果有两个输入点,那么如果两个点重合,则删除其中一个点并返回。
- 计算锚点,并删除与其重合的所有点(代码段12-2中的方法`anchorPointSearchAnd-Remove`)。如果没有剩下点或只剩下一个点,则重新插入锚点并返回。
- 如果未出现上述任何一种特殊情况,即至少剩下两个点,则利用方法`sortPoints`(代码段12-2)沿着锚点的逆时针方向对点排序,并传递一个`ConvexHullComparator`(它是一个比较器,允许一般排序算法沿着点的径向逆时针方向对点排序,如算法12-6中所需要的那样)。
- 准备进行Graham扫描,除了距锚点最远的那个点之外,删除排序表中任何初始共线的点,(代码段12-3中的方法`removeInitialIntermediatePoints`)。
- 调用代码段12-3中的方法`scan`,执行算法的扫描阶段。

总之,在实现计算几何算法时,必须考虑特殊情况,处理所有可能的“退化”情形。

583

代码段12-1 Graham扫描算法的Java实现中的方法`grahamScan`

```
public class ConvexHull {
    private static Sequence hull;
    private static Point2D anchorPoint;
    private static GeomTester2D geomTester = new GeomTester2DImpl();
    // public class method
    public static Sequence grahamScan (Sequence points) {
        Point2D p1, p2;
        copyInputPoints(points); // copy into hull the sequence of input points
        switch (hull.size()) {
            case 0: case 1:
                return hull;
```

```

    case 2:
        p1 = (Point2D)hull.first().element();
        p2 = (Point2D)hull.last().element();
        if (geomTester.areEqual(p1,p2))
            hull.remove(hull.last());
        return hull;
    default: // at least 3 input points
        // compute anchor point and remove it together with coincident points
        anchorPointSearchAndRemove();
        switch (hull.size()) {
            case 0: case 1:
                hull.insertFirst(anchorPoint);
                return hull;
            default: // at least 2 input points left besides the anchor point
                sortPoints(); // sort the points in hull around the anchor point
                // remove the (possible) initial collinear points in hull except the
                // farthest one from the anchor point
                removeInitialIntermediatePoints();
                if (hull.size() == 1)
                    hull.insertFirst(anchorPoint);
                else { // insert the anchor point as first and last element in hull
                    hull.insertFirst(anchorPoint);
                    hull.insertLast(anchorPoint);
                    scan(); // Graham's scan
                    // remove one of the two copies of the anchor point from hull
                    hull.remove(hull.last());
                }
                return hull;
        }
    }
}
}

```

584

代码段12-2 代码段12-1中的方法grahamScan调用的辅助方法copyInputPoints、
anchor PointSearchAndRemove和sortPoints

```

private static void copyInputPoints (Sequence points) {
    // copy into hull the sequence of input points
    hull = new NodeSequence();
    Enumeration pe = points.elements();
    while (pe.hasMoreElements()) {
        Point2D p = (Point2D)pe.nextElement();
        hull.insertLast(p);
    }
}

private static void anchorPointSearchAndRemove () {
    // compute the anchor point and remove it from hull together with
    // all the coincident points
    Enumeration pe = hull.positions();
    Position anchor = (Position)pe.nextElement();
    anchorPoint = (Point2D)anchor.element();
    // hull contains at least three elements
    while (pe.hasMoreElements()) {
        Position pos = (Position)pe.nextElement();
        Point2D p = (Point2D)pos.element();
        int aboveBelow = geomTester.aboveBelow(anchorPoint,p);
        int leftRight = geomTester.leftRight(anchorPoint,p);
        if (aboveBelow == GeomTester2D.BELOW ||

```

```

        aboveBelow == GeomTester2D.ON &&
        leftRight == GeomTester2D.LEFT) {
            anchor = pos;
            anchorPoint = p;
        }
        else
            if (aboveBelow == GeomTester2D.ON &&
                leftRight == GeomTester2D.ON)
                hull.remove(pos);
    }
    hull.remove(anchor);
}
private static void sortPoints() {
    // sort the points in hull around the anchor point
    SortObject sorter = new ListMergeSort();
    ConvexHullComparator comp = new ConvexHullComparator(anchorPoint,
                                                         geomTester);

    sorter.sort(hull, comp);
}

```

585

代码段12-3 代码段12-1中的方法grahamScan调用的辅助方法removeInitialIntermediatePoints和scan

```

private static void removeInitialIntermediatePoints() {
    // remove the (possible) initial collinear points in hull except the
    // farthest one from the anchor point
    boolean collinear = true;
    while (hull.size() > 1 && collinear) {
        Position pos1 = hull.first();
        Position pos2 = hull.after(pos1);
        Point2D p1 = (Point2D)pos1.element();
        Point2D p2 = (Point2D)pos2.element();
        if (geomTester.leftRightTurn(anchorPoint, p1, p2) ==
            GeomTester2D.COLLINEAR)
            if (geomTester.closest(anchorPoint, p1, p2) == p1)
                hull.remove(pos1);
            else
                hull.remove(pos2);
        else
            collinear = false;
    }
}
private static void scan() {
    // Graham's scan
    Position first = hull.first();
    Position last = hull.last();
    Position prev = hull.first();
    Position curr = hull.after(prev);
    do {
        Position next = hull.after(curr);
        Point2D prevPoint = (Point2D)prev.element();
        Point2D currPoint = (Point2D)curr.element();
        Point2D nextPoint = (Point2D)next.element();
        if (geomTester.leftRightTurn(prevPoint, currPoint, nextPoint) ==
            GeomTester2D.LEFT_TURN)
            prev = curr;
        else {
            hull.remove(curr);
        }
    }
}

```

```

        prev = hull.before(prev);
    }
    curr = hull.after(prev);
}
while (curr != last);
}
}

```

586

12.7 习题

基础题

- R-12.1 扩展对算法12-1中算法1DTreeRangeSeach运行时间的分析,使之适合二叉查找树 T 中包含 k_1 和/或 k_2 的情况。
- R-12.2 验证函数 $\Delta(p_1, p_2, p_3)$ 的绝对值是平面上点 p_1 、 p_2 和 p_3 所形成的三角形面积的两倍。
- R-12.3 给出定理12.5的完整证明。
- R-12.4 给出定理12.6的完整证明。
- R-12.5 给出示例12.1的完整证明。
- R-12.6 给出定理12.7的完整证明。
- R-12.7 给出定理12.8的完整证明。
- R-12.8 如果范围树的主结构不要求具有 $O(\log n)$ 的高度,那么范围树最坏情况下的空间复杂度是多少?
- R-12.9 给定一棵按照 n 个对象集合的 x 坐标构建的二叉查找树 T ,描述一个 $O(n)$ 时间的方法,计算 T 中每个结点 v 的 $\min_x(v)$ 和 $\max_x(v)$ 。
- R-12.10 证明优先查找树中的 high_y 值满足堆序性质。
- R-12.11 论证用优先查找树解答三边范围查找查询的算法是正确的。
- R-12.12 对于定义在平面上 n 个点上的 k -d树,它在最坏情况下的深度是多少?在高维情况下又是多少?
- R-12.13 假设集合 S 包含 n 个二维点,它的坐标都是 $[0, M]$ 区间内的整数。定义在 S 上的四叉树最坏情况下的深度是多少?
- R-12.14 画出以下点集的一棵四叉树,假设边界区域为 16×16 :

$\{(1, 2), (4, 10), (14, 3), (6, 6), (3, 15), (2, 2), (3, 12), (9, 4), (12, 14)\}$

- R-12.15 构造习题R-12.14中点集的一棵 k -d树。

587

- R-12.16 构造习题R-12.14中点集的一棵优先查找树。

创新题

- C-12.1 在二维范围树中使用的 $\min_x(v)$ 和 $\max_x(v)$ 标记不是严格需要的。描述一个在二维范围树中进行二维范围查找查询的算法,主结构中的每个内部结点只存储 $\text{key}(v)$ 标记(它是其元素的 x 坐标),你所设计方法的运行时间是多少?
- C-12.2 给出构造平面上 n 个点集的范围树的算法的伪代码描述,要求算法的运行时间为 $O(n \log n)$ 。
- C-12.3 描述存储有序关键字的 n 个数据项的集合 S 的有效数据结构,使它支持 $\text{rankRange}(a, b)$ 方法,枚举出其关键字的位序在范围 $[a, b]$ 中的所有数据项,其中 a 和 b 是区间 $[0, n-1]$ 中的整数。描述对象插入方法和删除方法,并表征这些方法和 rankRange 方法的运行时间。
- C-12.4 设计一种存储 n 个二维点集 S 的静态数据结构(不支持插入和删除操作),并能用 $O(\log^2 n)$ 的时间解答形如 $\text{countAllInRange}(a, b, c, d)$ 的查询,返回 S 中 x 坐标在范围 $[a, b]$ 内、 y 坐标在范围 $[c, d]$ 内的点数。这种结构所用的空间是多少?
- C-12.5 设计一种用 $O(\log n)$ 的时间解答 countAllInRange 查询(如上一个习题所定义的)的数据结构。
提示:考虑在每个结点上存储辅助结构,它们被“连接到”近邻结点的结构上。

C-12.6 证明如何扩展二维范围树, 从而对于 d 维点集, 可用 $O(\log^d n)$ 的时间解答 d 维范围查找查询问题, 其中常数 $d \geq 2$ 。

提示: 设计一个递归数据结构, 它利用 $(d-1)$ 维结构构建 d 维结构。

C-12.7 给定一个范围查找数据结构 D , 可以解答 d 维空间上 n 个点集的范围查找查询, 其中 d 为固定维数(如8、10或20), 查询时间为 $O(\log^d n + k)$, 其中 k 是答复数量。证明对于平面上的 n 个矩形集合 S , 如何利用 D 解答以下查询:

- `findAllContaining(x, y)`: 返回 S 中包含点 (x, y) 的所有矩形的枚举。
- `findAllIntersecting(a, b, c, d)`: 返回与矩形(其 x 范围为 $[a, b]$, y 范围为 $[c, d]$)相交的所有矩形的枚举。

解答每个这样的查询所需的运行时间是多少?

C-12.8 设 S 是形如区间 $[a, b]$ 的集合, 其中 $a < b$ 。设计一个有效数据结构, 回答形如`contains(x)`的查询, 所需时间为 $O(\log n + k)$ 。要求枚举出 S 上包含 x 的所有区间, 其中 k 是这些区间的个数。你所设计的数据结构要使用多大的空间?

588

C-12.9 描述一个有效的方法, 把一个对象插入一棵(平衡)优先查找树中。这个方法的运行时间是多少?

C-12.10 给定 n 个不相交线段 s_0, s_1, \dots, s_{n-1} 的基于数组的序列 S , 并且它们的端点在直线 $y = 0$ 和 $y = 1$ 上, 并从左至右排序。给定一个满足 $0 < y(q) < 1$ 的点 q , 设计一个算法, 要求用 $O(\log n)$ 的时间计算 S 中位于 q 右边的线段 s_i , 或报告 q 位于所有线段的右边。

C-12.11 给出一个 $O(n)$ 时间的算法, 测试一个点 q 是否在 n 个顶点的不相交多边形 P 的内部、外部或者边界上。当 q 的 y 坐标等于 P 中一个或多个顶点的 y 坐标时, 你的算法也能正确工作。

C-12.12 设计一个 $O(n)$ 时间的算法, 测试给定的 n 个顶点的多边形是否是凸多边形。不假设 P 是不相交的多边形。

C-12.13 设 S 是线段集合。给出一个算法, 确定 S 中的线段是否构成一个多边形。允许多边形是相交多边形, 但不允许多边形的两个顶点重合。

C-12.14 设 S 是平面上的 n 条线段集合。给出一个 $O((n+k)\log n)$ 时间的算法, 枚举出 S 中的 k 对相交的线段。

提示: 使用平面扫描技术, 包括作为事件的线段相交。注意: 你无法提前知道这些事件, 但是在扫描时总有可能知道下一个要处理的事件。

C-12.15 设计一个凸多边形的数据结构, 它利用线性空间, 且支持对数时间的点包含测试。

C-12.16 给定 n 个点的集合 P , 设计一个构造不相交多边形的有效算法, 其顶点是 P 中的点。

C-12.17 设计一个 $O(n^2)$ 时间的算法, 测试 n 个顶点的多边形是否是不相交的。假设按照其顶点的列表给出多边形。

C-12.18 给出简化的Graham扫描算法(算法12-6)输入点配置的例子, 使得算法不能正确工作。

C-12.19 设 P 是平面上的 n 个点集。修改Graham扫描算法, 使之对于 P 中不是凸包顶点的每个顶点 p , 计算 P 中顶点构成的三角形, 或者计算端点在 P 中且包含 p 于其内部的一条线段。

C-12.20 给定平面上的 n 个点集 S , 定义 S 的Voronoi图(Voronoi diagram)是区域 $V(p)$ 的集合, 称 $V(p)$ 为Voronoi单元(Voronoi cell)。对于 S 中的每个点 p , 定义 $V(p)$ 为平面上所有满足 p 是 S 中 q 的最近邻的所有点 q 的集合。

a. 证明Voronoi图中的每个单元都是凸单元。

b. 证明如果 p 和 q 是集合 S 中的最近点对, 那么Voronoi单元 $V(p)$ 和 $V(q)$ 接触。

c. 证明当且仅当 p 的Voronoi单元 $V(p)$ 是无界的时, 点 p 在集合 S 凸包的边界上。

589

C-12.21 给定平面上的点集 S , 定义 S 的Delaunay三角剖分(Delaunay triangulation)是所有三角形 (p, q, r) 的集合, 满足 p, q 和 r 在 S 中, 以这些点在边界上所定义的圆为空——圆内部不含 S 中的点。

a. 证明如果 p 和 q 是集合 S 中的最近点对, 那么 p 和 q 被Delaunay三角剖分中的一条边相连。

b. 证明当且仅当 p 和 q 被 S 的Delaunay三角剖分中的一条边相连时, Voronoi单元 $V(p)$ 和 $V(q)$ 共享点集 S 的Voronoi图中的一条边。

程序设计

- P-12.1 制作计算点集凸包的礼品包扎算法和Graham扫描算法的一个动画。
- P-12.2 用范围树数据结构或者优先范围树数据结构实现一个支持范围查找查询的类。
- P-12.3 实现四叉树数据结构和 k - d 树数据结构, 并进行实验性研究, 比较它们在范围查找查询方面的性能。

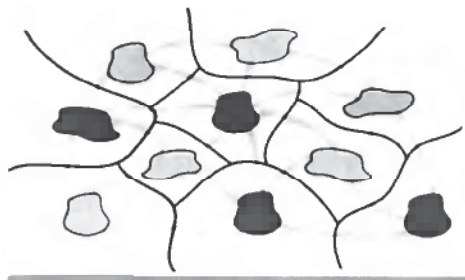
12.8 本章注记

本章介绍的凸包算法是Graham[88]给出的算法的一种变体。我们给出的求正交线段相交的平面扫描算法由Bentley和Ottmann[31]提出。给出的最近点算法结合了Bentley[28]和Hinrichs等人[94]的思想。

有几本计算几何方面的优秀著作, 包括Edelsbrunner[63]、Mehlhorn[150]、O'Rourke[160]、Preparata和Shamos[168]的著作, 以及Goodman和O'Rourke[83]、Pach[162]编辑的手册。其他进一步的阅读材料包括Aurenhammer[17]、Lee和Preparata[129]的综述性论文, 以及Goodrich[84]、Lee[128]和Yao[212]著作中的部分章节。此外, Sedgewick[182, 183]的著作包含了关于计算几何的若干章, 其中有非常好看的图片。实际上, Sedgewick著作中的图片激发了我们给出本书中的许多图片。

Mehlhorn[150]、Samet[175, 176]和Wood[211]的著作中讨论了多维范围查找树。请参阅这些书, 对多维查找树的历史展开讨论, 包括求解范围查找问题的各种数据结构。优先查找树归因于McCreight[140], 尽管Vuillemin[208]以“笛卡儿树”这个名称更早地介绍了这种结构。在McCreight[140]以及Aragon和Seidel[12]的描述中, 它们也称为“treap”。Edelsbrunner[62]说明了如何用它求解近似范围查找问题[15]。对范围查找数据结构的最近进展感兴趣的读者可以参考Agarwal[3, 4]的著作或Matoušek[138]的调查论文。Luca Vismara开发了12.6节中给出的凸包算法的实现。

第13章



NP 完全性

某些计算问题是困难的。我们绞尽脑汁想要找到求解它们的有效算法。但我们一次又一次地经历失败。在这些情况下，如果能够证明不可能找出一个有效算法也是好的。当有效算法规避我们时，这样一个证明也会大大减轻痛苦，这样我们也会由于问题不存在有效算法而感到安慰。不幸的是，这种证明通常甚至比提出有效算法更困难。

然而，并非所有的问题都会使人受挫。本章讨论的一些主题让我们知道某些问题确实是在计算上有难度的。其证明涉及称为NP完全性（NP-completeness）的概念。这个概念使我们可以严格证明找出某个问题的一个有效算法至少和找出一大类称为“NP”的问题中的所有问题的有效算法一样困难。这里使用的“有效”的形式概念是问题具有的算法的运行时间与其输入大小 n 的多项式函数成正比（回忆这种效率的概念已在1.2.2节中提到过）。即如果一个算法的运行时间为 $O(n^k)$ ，则认为它是“有效的”，其中 n 为输入大小， $k > 0$ 为常数。即便如此，NP类中包含一些极其难的问题，数十年来，研究人员还未找出它们的多项式时间解。于是，证明一个问题是NP完全的与证明问题不可能存在有效算法截然不同。它仍然是一种强大的陈述。实质上讲，证明一个问题 L 是NP完全的，也就是说尽管我们不能找到 L 的一个有效算法，任何计算机科学家也还没有找出这样的算法。的确，大多数计算机科学家坚持认为在多项式时间内不可能解决任何NP完全问题。

在这一章里，形式化地定义了NP类及其相关类P，以及说明如何证明一个问题是NP完全的。还讨论了一些广为人知的NP完全问题。证明了每个这样的问题至少和NP中的其他问题一样困难。这些问题包括可满足性问题、顶点覆盖问题、KNAPSACK问题和旅行推销员问题。

然而，我们并没有停在那里。因为许多这样的问题相当重要，这在于它们和优化问题紧密相关，在现实世界中它们的解时常对应节约金钱、时间或其他资源。因此，这一章还讨论了处理NP完全性的一些方法。其中最有效的一种方法是构造NP完全问题的多项式时间的近似算法。尽管这样的算法常常不会产生最优解，但是它们时常接近于最优。事实上，在某种情况下，可以保证一个近似算法与最优解相近的程度。这一章探讨几种这样的情况。

本章最后包括一些技术，它们在解决实际NP完全问题中非常有效。尤其是介绍的回溯法（backtracking）和分枝限界法（branch-and-bound）。用它们构造的算法最坏情况下的运行时间为指数时间，但利用某些情形也有可能得到更快的算法。我们给出了两种技术的Java例子。

13.1 P 类和 NP 类

为了研究NP完全性问题，需要更准确地定义运行时间。也就是说，我们不是把输入大小非形式地表示为“构成”输入的数据项数（见第1章），而是把问题的输入大小（input size） n 定义为编码输入实例所用的位数。还假设输入中的字符和数字可用合理的二进制编码模式编码，使得用常数个位就可表示每个字符，至多用 $c \log M$ 位就可表示每个大于0的整数 M ，其中常数 $c > 0$ 为常数。特别是不允许一元编码（unary encoding），例如用 M 个1表示整数 M 。

回忆我们已经在本书余下部分中定义 n 为输入中的“数据项”数。但是，现在暂时用 N 表示输入中的数据项数，用 n 表示对输入进行编码的位数。因此，如果 M 是输入中的最大整数，那么 $N + \log M \leq n \leq cN \log M$ ， $c > 0$ 为常数。形式上讲，我们把算法 A 最坏情况下的运行时间（running time）定义为算法 A 最坏情况下所花费的时间，它是 n 的函数，取自编码为 n 位的所有可能的输入。幸运的是，正如在以下引理中所看到的，大多数运行时间为 N 的多项式时间的算法仍会导致运行时间为 n 的多项式时间的算法。如果对于用 b 位表示的一个对象或两个对象的基本操作，其结果对象至多用 $b + c$ 位表示，其中 $c \geq 0$ ，则定义算法是 c 增量的（ c -incremental）。例如，对于任何常数 c ，利用乘法作为基本操作的算法可能不是 c 增量的算法。当然可以在 c 增量的算法中包含可以进行乘法操作的例程，但这里不应该把这个例程统计为基本操作。

引理13.1 在RAM模型中，如果一个 c 增量的算法 A 最坏情况下的运行时间为输入项数 N 的函数 $t(N)$ ，其中常数 $c > 0$ ，那么算法 A 的运行时间可表示为 n 的函数 $O(n^2 t(n))$ ，其中 n 为输入的标准非一元编码串的位数。

证明 注意 $N \leq n$ 。因此， $t(N) \leq t(n)$ 。同样，如果算法 A 中包括用 $b \geq 1$ 位表示的一个对象或两个对象，因为 c 是常数，那么执行其中每个基本操作至多利用 db^2 个位操作，其中常数 $d \geq 1$ 。这些基本运算包括所有比较、控制流程和基本的非乘法算术操作。然而，在 c 增量算法的 N 个步骤中，表示任一对象所用的最大位数为 $cN + b$ ，其中 b 是输入对象的最大规模。但是， $cN + b \leq (c+1)n$ 。因此，完成 A 中的每一步至多需要 $O(n^2)$ 位的步骤。 ■

于是，任一运行时间是输入数据项数的多项式时间的“合理”算法，其运行时间也是输入位数的多项式时间。因此，对于本章其余部分，我们用 n 作为问题的输入大小和“数据项”数，并且理解任何多项式时间的算法的运行时间必定为输入位数的多项式时间。

593

13.1.1 定义复杂类 P 和复杂类 NP

对于本书中讨论过的一些问题，如图问题、文本处理或排序问题，由引理13.1可知，这些问题曾经具有的多项式时间算法可以转换成位模型的多项式时间算法。甚至对于计算整数 x 的幂的反复平方算法（10.1.4节），如果用它来把 x 的值提高到一个 $O(\log n)$ 位表示的数，其运行时间也为操作数的多项式。因而，多项式时间是衡量问题难易性的一个有用概念。

而且，多项式类对于加法、乘法和组合运算是封闭的。也就是说，如果 $p(n)$ 和 $q(n)$ 是多项式，那么 $p(n) + q(n)$ 、 $p(n) \cdot q(n)$ 和 $p(q(n))$ 也是多项式。因此，我们可以通过组合或组成多项式时间的算法，来构造新的多项式时间的算法。

1. 判定问题

为了简化讨论，暂时讨论判定问题（decision problem），即希望输出为“yes”或者“no”的计算问题。换句话说，判定问题的输出只有一位，为0或1。例如，下列每个问题都是判定问题。

- 给定串 T 和串 P , P 是否是 T 的子串?
- 给定两个集合 S 和 T , S 和 T 是否包含相同的元素集合?
- 给定具有正权值边的图 G 及一个整数 k , 试问 G 中存在权值至多为 k 的最小生成树吗?

事实上, 最后一个问题表明, 当我们试图最小化或者最大化优化问题中的某些值时, 可将这个优化问题 (optimization problem) 转换成一个判定问题, 即我们可以引入参数 k , 并询问优化问题的最优值是否至多为 k 或者至少为 k 。注意, 如果我们能够证明判定问题是困难的, 那么其相应的优化问题也必定是困难的。

2. 问题和语言

如果输入为 x 的算法 A 输出“yes”, 就称算法 A 接受 (accept) 输入串 x 。因而, 可以把判定问题实际上只看作串的集合 L ——即正确解决那个问题的算法所能接受的串。因为常常称串的集合为语言 (language), 所以用字母“ L ”表示判定问题。我们可以进一步扩展这个基于语言的观点, 如果对于 L 中的每个 x , 算法 A 输出“yes”, 其他算法则输出“no”, 则称算法 A 接受语言 L 。在本章中, 假设如果 x 是一个不正确的语法, 那么给定 x 的算法输出“no” (注意: 某些课本上允许算法 A 有进入无限循环的可能性, 对于某些输入不产生任何输出, 但本书中只关注计算在有限步骤内终止的算法)。

594

3. 复杂类P

复杂类P (complexity class P) 是指最坏情况下在多项式时间内可以接受的所有判定问题 (或者语言) L 的集合。也就是存在算法 A , 如果 $x \in L$, 那么在输入 x 上, A 在 $p(n)$ 时间内输出“yes”, 其中 n 是 x 的大小, $p(n)$ 是多项式。这里需要注意的是, P的定义并没有表明拒绝一个输入所需的运行时间——当算法 A 输出“no”时。这种情况是指语言 L 的补 (complement), 它由不在 L 中的所有二进制串组成。给定多项式时间 $p(n)$ 接受语言 L 的算法 A , 我们仍然可以容易地构造一个接受 L 补的多项式时间的算法。尤其是给定输入 x , 可以构造一个补算法 B , 它简单地运行 $p(n)$ 步的算法 A , 其中 n 是 x 的规模, 并且如果算法 B 试图运行多于 $p(n)$ 步, 就会终止算法 A 。如果 A 输出“yes”, 那么 B 输出“no”。同样, 如果 A 输出“no”, 或者 A 运行至少 $p(n)$ 步而没有产生任何输出, 那么 B 输出“yes”。无论哪一种情况, 补算法 B 的运行时间均为多项式时间。因此, 如果表示某个判定问题的语言 L 是在P中, 那么 L 的补也在P中。

4. 复杂类NP

复杂类NP (complexity class NP) 的定义不但包括复杂类P, 还可能包括不在P中的语言。确切地讲, 对于NP类问题, 允许算法执行另一个操作:

- **choose(b):** 这个操作以非确定性的方式选择一位 (即为0或1), 并将选择的值赋予 b 。

当算法 A 中利用了choose基本操作时, 称算法 A 是非确定性的 (nondeterministic)。如果存在调用choose的结果集, 使得算法 A 对于输入串 x 最终会输出“yes”, 则称算法 A 非确定性地接受一个串 x 。换句话说, 就好像是我们考虑调用choose后产生的所有可能结果, 而只选择了那些导致接受的结果, 如果存在这样一个结果集。注意这不同于随机选择。

复杂类NP就是那些在多项式时间内被非确定性地接受的判定问题 (或者语言) L 的集合, 即存在非确定性算法 A , 如果 $x \in L$, 那么在输入 x 上, 在 A 中存在choose调用的结果集, 使得 A 在 $p(n)$ 时间内输出“yes”, 其中 n 是 x 的大小, $p(n)$ 是多项式。注意NP的定义并没有表明拒绝的运行时间。实际上, 允许在多项式时间 $p(n)$ 内接受语言 L 的算法 A 输出“no”时所花费的时间比 $p(n)$ 步要多。此外, 由于非确定性接受可能涉及在多项式级调用choose方法, 因而如果一种语言 L 在NP中, 那么 L 的补未必也在NP中。实际上, 存在一个称为co-NP的复杂类, 包括了其补在NP中的所有语言, 许多研究人员认为co-NP \neq NP。

595

5. NP的另一种定义

实际上存在复杂类NP的另一种定义, 这个定义对于某些读者更直观。NP的这个定义基于确定性验证, 而不是非确定性接受。给定 L 中的任意串 x 作为输入, 如果存在另一个串 y , 满足对于输入 $z = x + y$, 算法 A 输出“yes”, 则称语言 L 可被算法 A 验证 (verified), 其中使用符号“+”表示连接。因为串 y 帮助我们证明 x 的确在 L 中, 称串 y 为 L 中成员的证书 (certificate)。注意当一个串不在 L 中时, 我们不做关于验证的声明。

验证概念可使我们给出复杂类NP的另一种定义, 即把复杂类NP定义为所有语言 L 的集合, 这些语言 L 定义的判定问题可在多项式时间内得到验证。也就是说, 存在(确定性)算法 A , 对于 L 中的任一 x , 利用证书 y 在多项式时间 $p(n)$ 内验证 x 的确在 L 中, $p(n)$ 包括算法 A 读其输入 $z = x + y$ 的时间, 其中 n 是 x 的大小。注意这个定义蕴涵着 y 的大小小于 $p(n)$ 。正如以下定理所表明的那样, 基于验证的NP定义与上述给出的基于非确定性的NP定义是等价的。

定理13.1 当且仅当 L 可在多项式时间内被非确定性地接受时, 语言 L 可在多项式时间内被(确定性)地验证。

证明 考虑每一种可能性。首先假定 L 可在多项式时间内被验证, 即存在确定性算法 A (不进行choose调用), 对于给定多项式长度的证书 y , 可在多项式时间 $p(n)$ 内验证串 x 在 L 中。于是, 可以构造以串 x 作为输入的非确定性算法 B , 并调用choose方法, 对 y 中的每一位赋值。给定证书 y , 在 B 完成串 $z = x + y$ 的构造之后, 然后调用算法 A 来验证 $x \in L$ 。如果存在证书 y , 满足 A 接受 z , 那么显然存在 B 的非确定性选择集合, 导致 B 自身输出“yes”, 且 B 运行 $O(p(n))$ 步。

下面, 假定 L 可在多项式时间内被非确定性地接受, 即给定 L 中的串 x , 存在执行 $p(n)$ 步的非确定性算法 A , 可能包括那些choose步骤, 对于这些choose步骤的某个结果序列, 满足 A 输出“yes”。给定 L 中的串 x , 存在一个确定性验证算法 B , 利用输入为 x 的算法 A 调用choose产生的所有结果的有序连接作为它的证书 y , 最终产生输出“yes”。因为算法 A 运行 $p(n)$ 步, 对于给定的输入 $z = x + y$, 算法 B 也运行 $O(p(n))$ 步, 其中 n 是 x 的大小。 ■

这个定理实际上蕴涵着, 这两种NP的定义是等价的, 我们可以利用任何一种定义来证明一个问题在NP中。

596

6. P = NP问题

计算机科学家并不能够肯定是否 $P = NP$ 。研究人员甚至不确信是否 $P = NP \cap \text{co-NP}$ 。尽管如此, 大多数科学家认为 P 与 NP 和 co-NP 都不相等, 也不与它们的交集相等。事实上, 在以下讨论的NP问题的例子中, 许多人认为它们并不在 P 中。

13.1.2 NP 中的一些有趣问题

对定理13.1的另一种解释是, 它蕴涵着我们总能够构造一个非确定性算法, 使得它首先执行其所有choose步骤, 算法的余下部分只进行验证。在这一节里通过几个例子说明证明有趣的判定问题在NP中的方法。第一个例子是关于图的问题。

HAMILTONIAN-CYCLE(哈密尔顿回路)问题是指: 以图 G 作为输入, 询问 G 中是否存在简单的回路, 它只访问 G 中的每个顶点一次, 并回到它的起始顶点。称这样的回路为 G 的哈密尔顿回路。

引理13.2 HAMILTONIAN-CYCLE在NP中。

证明 定义非确定性算法 A , 以编码为二进制表示的邻接表的图 G 作为输入, 从1到 N 对顶点编号。定义 A 首先反复调用choose过程, 确定1~ N 之间的 $N+1$ 个数的序列 S , 然后除了 S 中的第一

个数和最后一个数（它们应该是相同的）之外， A 会检查 S 中从1到 N 的其他每个数刚好出现一次（例如，通过对 S 排序）。然后验证序列 S 定义了 G 中顶点和边的一个回路。显然序列 S 的二进制编码大小至多为 n ，其中 n 为输入大小。而且，对序列 S 所做的两次检查均可在 n 的多项式时间内完成。

观察可得，如果存在 G 的一个回路，访问 G 中每个顶点一次，并回到它的起始顶点，那么存在使 A 输出“yes”的序列 S 。同样，如果算法 A 输出“yes”，那么它已经找到 G 中的一个回路，访问 G 中的每个顶点一次，并回到它的起始顶点，即 A 非确定性地接受了语言HAMILTONIAN-CYCLE。换句话说，HAMILTONIAN-CYCLE在NP中。 ■

下一个例子关于电路设计测试问题。布尔电路（Boolean circuit）是一个有向图，图中的每个顶点称为逻辑门，对应简单的布尔函数AND、OR或NOT。逻辑门的入边是其对应布尔函数的输入，出边是其对应布尔函数的输出，对于那个门而言，都将具有相同的值（见图13-1）。没有入边的顶点为输入结点，没有出边的顶点为输出结点。

597

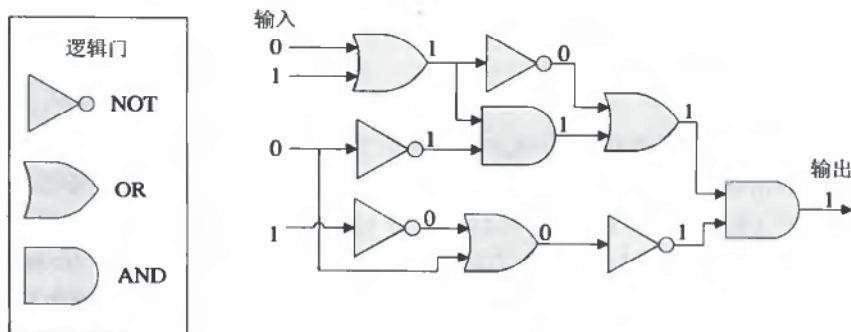


图13-1 布尔电路示例

CIRCUIT-SAT（电路可满足性）问题是指：取只有一个输出结点的布尔电路作为输入，问是否存在电路输入的一种指派值，满足它的输出值为“1”。这样的指派值称为满足指派（satisfying assignment）。

引理13.3 CIRCUIT-SAT在NP中。

证明 我们构造一个在多项式时间内接受CIRCUIT-SAT的非确定性算法，首先利用choose方法去“猜测”输入结点的值以及每个逻辑门的输出值，然后简单访问电路 C 中的每个逻辑门 g ，即访问至少有一条输入边的每个顶点。根据给定的 g 的输入值，检查所猜测的 g 的输出值事实上是 g 所对应布尔函数的正确值，布尔函数可以是AND、OR或NOT。这个计算过程可以容易地用多项式时间完成。如果对任一逻辑门的检查失败，或者说所猜测的输出值为0，那么算法输出“no”。另一方面，如果对于每个逻辑门的检查都成功，并输出“1”，那么算法输出“yes”。因此，如果确实存在 C 的输入值满足指派，那么存在choose语句的可能结果集，使得算法在多项式时间输出“yes”。同样，如果存在choose语句的结果集，使得算法在多项式时间输出“yes”，那么必定存在 C 的输入值满足指派。于是，CIRCUIT-SAT在NP中。 ■

下一个例子说明了如何证明一个优化问题的判定问题是在NP中。给定图 G ，图 G 的顶点覆盖（vertex cover）是顶点的一个子集 C ，满足对于 G 中的每条边 (v, w) ， $v \in C$ 或者 $w \in C$ （可能两者都在 C 中），优化的目标是找出尽可能小的 G 的顶点覆盖。

598

VERTEX-COVER (顶点覆盖) 问题的判定问题是指: 取图 G 和整数 k 作为输入, 询问是否存在至多包含 k 个顶点的 G 的顶点覆盖。

引理13.4 VERTEX-COVER在NP中。

证明 给定整数 k 和图 G 。 G 中顶点从1到 N 进行编号。我们可以反复调用choose方法, 构造 $1 \sim N$ 之间的 k 个数的集合 C 。作为一个验证, 把 C 中的所有数插入一个字典中, 然后检查 G 中的每条边, 证实对于 G 中的每条边 (v, w) , 有 v 在 C 中, 或者 w 在 C 中。如果找到一条边, 它的两个端点都不在 G 中, 那么输出“no”。如果检查了 G 中所有的边, 满足每条边都有一个顶点在 C 中, 那么输出“yes”。显然, 这样的计算运行时间为多项式时间。

注意如果存在 G 的至多为 k 的顶点覆盖, 那么存在定义集 C 的一种数值指派, 使得 G 中的每一条边通过所做的测试, 且算法输出“yes”。同样, 如果算法输出“yes”, 那么, 必定存在大小至多为 k 的顶点子集 C , 满足 C 是一个顶点覆盖。因此, VERTEX-COVER在NP中。 ■

给出了NP中的几个有趣的例子之后, 我们转到NP完全性概念的定义上。

13.2 NP 完全性

非确定性接受判定问题(或语言)的概念诚然有些人觉得陌生。毕竟, 不存在一台能够有效执行非确定算法的常规计算机, 可以多次调用choose方法。的确, 到目前为止还没有人证明, 一台非常规的计算机(如量子计算机或者DNA计算机)可以利用多项式数量级的资源, 来有效模拟非确定性多项式时间算法。可以肯定的一点是, 通过一步一步地对算法中调用choose语句产生的所有可能结果进行试验, 可以确定性地模拟一个非确定性算法。但是对于任何至少调用choose方法 n^ϵ 次的非确定性算法, 这项模拟的计算时间为指数时间, 其中 $\epsilon > 0$ 为固定常数。的确, 在复杂类NP中有数百个问题, 大多数计算机科学家坚持认为不存在解决这些问题的确定性多项式算法。

复杂类NP的用处在于, 它能够形式地捕获大量被认为是计算上很困难的问题。事实上, 存在某些问题, 经证明它们和NP中的其他问题一样困难, 我们关注问题的多项式时间的可计算性。难度的概念基于下面讨论的多项式归约的概念。

599

13.2.1 多项式时间归约和 NP 困难度

如果存在多项式时间内的可计算函数 f , 取 L 中的输入 x , 并把这个输入转换为 M 的一个输入 $f(x)$, 满足对于 $x \in L$, 当且仅当 $f(x) \in M$ 时, 称定义某个判定问题的语言 L 是多项式时间可归约 (polynomial-time reducible) 到语言 M 。此外, 用符号 $L \xrightarrow{\text{poly}} M$ 简洁地表示语言 L 可在多项式时间内归约到语言 M 。

如果NP中的每个其他语言 L 可在多项式时间内归约到 M , 则称定义某个判定问题的语言 M 是NP困难的 (NP-hard)。对于每个语言 $L \in \text{NP}$, 如果 $L \xrightarrow{\text{poly}} M$, 则从数学上表示为 M 是NP困难的。如果语言 M 是NP困难的, 且自身也在NP类中, 那么 M 是NP完全的 (NP-complete)。因此, 在非常正式的意义下, 当关注一个NP完全问题的多项式时间的可计算性时, 它是NP中最困难的问题之一。因为如果任何人能够证明, NP完全问题 L 在多项式时间内可解, 那么这直接蕴涵着整个NP类中的其他每个问题都在多项式时间内可解。在这种情况下, 可以通过把NP类中的其他NP语言 M 归约到语言 L , 然后运行语言 L 的算法, 来接受 M 。换句话说, 如果找到一个NP完全问题的确定性多项式时间算法, 那么 $P = \text{NP}$ 。

13.2.2 Cook-Levin 定理

首先, NP完全性的定义看上去过于严格。但是, 以下定理表明至少存在一个NP完全问题。

定理13.2 (Cook-Levin定理): CIRCUIT-SAT是NP完全问题。

证明 引理13.3证明了CIRCUIT-SAT在NP中。因此, 还需证明这个问题是NP困难的, 即要证明NP中的每个问题都可在多项式时间内归约到CIRCUIT-SAT。考虑NP中表示某个判定问题的语言 L 。给定多项式规模的证书 y , 由于 L 在NP中, 存在确定性算法 D , 在多项式时间 $p(n)$ 内接受 L 中的任意 x , 其中 n 是 x 的大小。证明的主要思想是构造一个较大但是具有多项式规模的电路 C , 模拟输入为 x 的算法 D , 按照这样一种方式, 当且仅当存在证书 y , 使得对于输入 $z = x + y$, 算法 D 输出“yes”时, C 是可满足的。

回忆(1.1.2节)任何一个确定性算法(如 D), 可用简单的计算模型(称为随机访问机或RAM)实现。它由一个CPU和一个可寻址的存储单元 M 组成。在我们的这种情况下, 存储单元 M 包括输入 x 、证书 y 、工作空间 W , 这些都是算法 D 进行计算和执行算法自身代码所需要的。 D 的工作空间 W 包括临时计算使用的所有寄存器, 以及算法 D 执行过程中调用所需的堆框架。 W 的最顶层的栈框架包含程序计数器(PC), 它用于识别算法 D 当前执行到程序中的位置。因此CPU自身没有存储单元。在执行 D 的每一步中, CPU读取PC指向的下一条指令 i , 进行 i 所指示的计算, 这些计算可以是比较、算术运算、条件跳转、过程调用的一步等, 然后更新PC, 使其指到下一条要执行的指令。因此, D 的当前状态完全由其存储单元中的内容表征。此外, 由于 D 在多项式 $p(n)$ 步内接受 L 中的 x , 其中 n 是 x 的大小。那么可以假设它的整个存储单元有效集只由 $p(n)$ 位组成, 这是因为在 $p(n)$ 步内, D 至多可以访问 $p(n)$ 个存储单元。还须注意的是, D 的代码规模是关于 x 、 y 甚至 W 大小的常数。我们把算法 D 一次执行中所需 $p(n)$ 规模的存储单元集 M 称为算法 D 的配置(configuration)。

600

把 L 归约到CIRCUIT-SAT问题的关键之处在于, 在我们的计算模型之下, 如何构造模拟CPU工作的布尔电路。构造的细节超出了本书的范围, 但众所周知, CPU可以被设计成只由AND、OR和NOT门组成的布尔电路。而且, 进一步认为, 包括与 $p(n)$ 位存储单元连接的地址单元的这条电路都可以被设计成以 D 的配置作为输入, 并把从处理下一个计算步骤产生的配置作为输出结果。此外, 我们称这条模拟电路为 S , 它至多可由 $cp(n)^2$ 个AND、OR和NOT门组成, 其中 $c > 0$ 为常数。

为了模拟 D 的整个 $p(n)$ 步, 我们制作 S 的 $p(n)$ 个副本, 使得一个副本的输出作为下一个副本的输入(见图13-2)。对 S 的第一个副本的输入中, 包括 D 程序的“硬线”值、 x 值、初始堆框架(连同指向 D 的第一条指令的PC)以及其余工作存储单元(都被初始化为0)。在第一个副本的输入中, 唯一未指明的真实输入是证书 y 的 D 的配置单元, 这些就是电路的真实输入。同样, 除了表明 D 的结果的一个输出之外, “1”代表“yes”, “0”代表“no”, 忽略了 S 最终副本的所有输出结果。电路 C 的总规模为 $O(p(n)^3)$, 它仍然是 x 大小的多项式。

在经过 $p(n)$ 步后, 考虑对于某些证书 y , D 所接受的一个输入 x 。此时, 存在对应于 y 的 C 输入的一个指派值, 通过 x 的这个输入指派和硬线值, C 模拟 D , 使得 C 最终输出“1”。因此, 在这种情况下 C 是可满足的。相反, 考虑 C 是可满足的情况, 存在与证书 y 对应的输入集, 满足 C 输出“1”。但因为 C 正确地模拟了算法 D , 这蕴涵着存在证书 y 的一个指派值, 满足 D 输出“yes”。因此, 在这种情况下, D 将验证 x 。于是, 当且仅当 C 是可满足时, D 接受证书为 y 的 x 。 ■

601

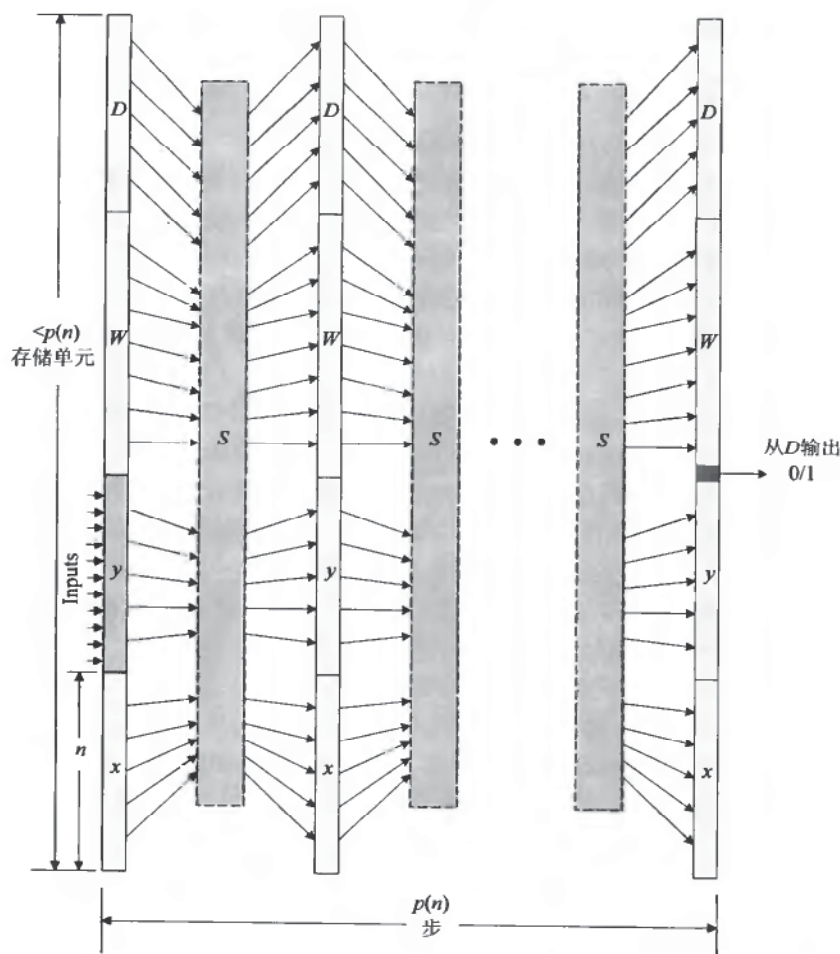


图13-2 证明CIRCUI-T-SAT是NP困难的电路说明。图中唯一的真实输入与证书 y 对应，问题实例 x 、工作存储器 W ，以及程序代码 D 初始化为“硬线”值。唯一的输出是确定算法是否接受 x 的位

602

13.3 重要的 NP 完全问题

定理13.2表明，确实存在NP完全问题，但即使我们走捷径，假定存在模拟电路 S ，证明这个事实仍然是一件累人的工作。值得庆幸的是，已经有一个问题被证明“彻头彻尾”是NP完全问题，我们可以利用简单多项式时间归约，来证明其他问题是NP完全问题。本节将要探究许多这样的归约。

只给定一个NP完全问题，我们现在可以利用多项式时间归约，来证明其他问题是NP完全问题。此外，我们会反复利用以下关于多项式归约的重要引理。

引理13.5 如果 $L_1 \xrightarrow{\text{poly}} L_2$ 且 $L_2 \xrightarrow{\text{poly}} L_3$ ，那么 $L_1 \xrightarrow{\text{poly}} L_3$ 。

证明 因为 $L_1 \xrightarrow{\text{poly}} L_2$ ， L_1 的任何实例 x 可在多项式时间 $p(n)$ 内被转换成 L_2 的实例 $f(x)$ ，满足当且仅当 $f(x) \in L_2$ 时， $x \in L_1$ ，其中 n 是 x 的大小。同样， $L_2 \xrightarrow{\text{poly}} L_3$ ， L_2 的任何实例 y 可在多项式时间 $q(m)$ 内被转换成 L_3 的实例 $g(y)$ ，满足当且仅当 $g(y) \in L_3$ 时， $y \in L_2$ ，其中 m 是 y 的大小。

将这两种构造组合起来, L_1 的任何实例 x 可在时间 $q(k)$ 内被转换成 L_3 的实例 $g(f(x))$, 满足当且仅当 $g(f(x)) \in L_3$ 时, $x \in L_1$, 其中 k 是 $f(x)$ 的大小。但由于在 $p(n)$ 步内构造了 $f(x)$, $k \leq p(n)$, 因此, $q(k) \leq q(p(n))$ 。因为两个多项式的组合总是得到另一个多项式, 这个不等式蕴涵着 $L_1 \xrightarrow{\text{poly}} L_3$ 。■

在这一节里, 利用引理证明了几个重要问题是NP完全问题。所有证明过程遵循一般框架。给定一个新的问题 L , 首先证明 L 是在NP中, 然后用多项式时间将一个已知的NP完全问题归约到 L , 证明 L 是NP困难的。因此, 证明了 L 是在NP中, 并且也是NP困难的; 因此, 也就证明了 L 是NP完全问题。(为什么不在另一个方向归约?) 这些归约通常具有以下三种形式之一:

- 限制法 (restriction): 通过说明一个已知的NP完全问题 M 实际上只是 L 的一个特例, 来证明问题 L 是NP困难的。
- 局部替换法 (local replacement): 将已知NP完全问题 M 的实例和 L 的实例划分成一些“基本单元”, 然后证明 M 的每个基本单元都可以局部地转换成为 L 的一个基本单元, 从而将一个已知的NP完全问题 M 归约到 L 。
- 分量设计法 (component design): 构造 L 实例的分量, 这些分量可以执行 M 实例的重要结构功能, 从而将一个已知的NP完全问题 M 归约到 L 。例如, 某些分量可能执行“选择”, 而其他分量执行函数“计算”。

上述三种形式中最后一种是最难构造的; 它也是在Cook-Levin定理(13.2)的证明中用过的形式。

603

在图13-3中, 说明了证明的问题是NP完全问题, 这些问题是由哪个问题归约而来, 以及在多项式归约中使用的技术。

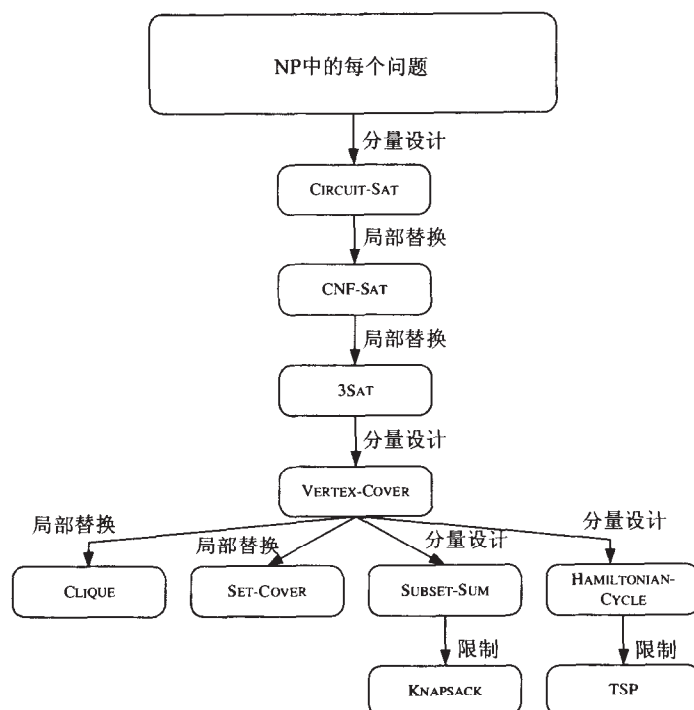


图13-3 几种基本NP完全性证明中使用的归约方法图示。每条有向边表示一个多项式时间的归约。边上的标记表示归约的主要形式。最顶层的归约是Cook-Levin定理

在本节的其余部分, 研究某些重要的NP完全问题, 并成对论述它们。每一对都强调一个重要的问题类, 包括涉及布尔公式、图、集合以及数中的问题。我们先从两个涉及布尔公式的问题开始。

604

13.3.1 CNF-SAT 和 3SAT

我们提出的第一个归约是涉及布尔公式的问题。布尔公式是一个括号化的表达式, 它由利用布尔操作(如OR (+)、AND (\cdot)、NOT (在子表达式上画一横条)、IMPLIES (\rightarrow) 以及IF-AND-ONLY-IF (\leftrightarrow)) 的布尔变量构成。如果布尔公式是由子表达式集通过AND组合而成, 则称布尔公式是合取范式(conjunctive normal form, CNF)。这些子表达式称为子句(clause), 子句由称为文字(literal)的布尔变量或者它们的非通过OR连接形成, 例如, 以下布尔公式是CNF:

$$(\bar{x}_1 + x_2 + x_4 + \bar{x}_7)(x_3 + \bar{x}_5)(\bar{x}_2 + x_4 + \bar{x}_6 + x_8)(x_1 + x_3 + x_5 + \bar{x}_8)$$

如果 x_2 、 x_3 和 x_4 的值为1, 这个公式的计算值为1, 用0表示false, 1表示true。称CNF为标准范式, 是因为任一布尔公式都可被转换成这种形式。

1. CNF-SAT

CNF-SAT问题取CNF形式的布尔公式作为输入, 询问是否存在对其布尔变量的一个指派值, 使得公式的计算值为1。

容易证明, CNF-SAT在NP中, 因为对于给定的布尔公式 S , 可以构造一个简单的非确定性算法, 首先猜测 S 中布尔变量的一个指派值, 然后依次计算 S 的每个子句, 如果 S 的所有子句值为1, 那么 S 是可满足的; 否则 S 是不可满足的。

为了证明CNF-SAT是NP困难的, 我们用多项式时间将CIRCUIT-SAT归约到这个问题。给定一条布尔电路 C , 不失一般性, 假设每个AND和OR门有两个输入, 每个NOT门有一个输入。为了构造与 C 等价的公式 S , 为整个电路 C 的每个输入构造一个变量 x_i , 并试图将变量集限制到这些变量 x_i 的集合上, 接着将这些输入的子表达式组合起来, 构造 C 的公式。但是, 一般而言, 这种方法并不会以多项式时间运行(见习题C-13.3)。而是对于 C 中每个门的输出构造一个变量 y_i 。然后, 按照下述方法, 构造 C 中每个门 g 的公式 B_g 。

- 如果 g 是一个输入为 a 和 b (可以是 x_i 或 y_i)、输出为 c 的AND门, 那么 $B_g = (c \leftrightarrow (a \cdot b))$ 。
- 如果 g 是一个输入为 a 和 b 、输出为 c 的OR门, 那么 $B_g = (c \leftrightarrow (a + b))$ 。
- 如果 g 是一个输入为 a 、输出为 b 的NOT门, 那么 $B_g = (b \leftrightarrow \bar{a})$ 。

希望通过所有 B_g 的AND来构成公式 S , 但是这样的公式可能不在CNF中。于是首先把每个 B_g 转换为CNF, 然后用AND操作将这些转换过的 B_g 组合起来, 定义CNF公式 S 。

605

为了把布尔公式 B 转换成CNF, 构造 B 的真值表, 如图13-4所示。然后对于表中求值为0每一行, 构造公式 D_i 。当且仅当在那一行中求值为0时, 每个 D_i 由真值表中变量及变量的非通过AND连接。将所有 D_i 通过OR组织起来构造公式 D , 由于公式 D 由子句通过OR连接而成, 而子句由变量及其非通过AND连接而成, 因此称这样的公式 D 为析取范式(disjunctive normal form, DNF)。在这种情况下, 我们得到等价于 \bar{B} 的DNF公式 D , 因为当且仅当 B 的值为0时, 它的值为1。为了将公式 D 转换成CNF形式的公式 B , 将De Morgan定律应用到它的每个 D_i 上, 有

$$\overline{(a+b)} = \bar{a} \cdot \bar{b} \text{ 和 } \overline{(a \cdot b)} = \bar{a} + \bar{b}$$

a	b	c	$B = (c \leftrightarrow (a \cdot b))$
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	0
0	0	0	1

$$\bar{B} \text{ 的 DNF 公式} = a \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot c + \bar{a} \cdot b \cdot c + \bar{a} \cdot \bar{b} \cdot \bar{c}$$

$$B \text{ 的 CNF 公式} = (\bar{a} + \bar{b} + c) \cdot (\bar{a} + b + \bar{c}) \cdot (a + \bar{b} + \bar{c}) \cdot (a + b + \bar{c}).$$

图13-4 布尔公式 B 的真值表, 布尔变量为 a 、 b 和 c 。DNF中 \bar{B} 的等价公式和CNF中 B 的等价公式

由图13-4可知, 我们可以用以下形式的CNF范式:

$$(\bar{a} + \bar{b} + c)(\bar{a} + b + \bar{c})(a + \bar{b} + \bar{c})(a + b + \bar{c})$$

替换形如 $(c \leftrightarrow (a \cdot b))$ 的每个 B_g 。同样, 我们可以用等价的CNF公式

$$(\bar{a} + \bar{b})(a + b)$$

替换形如 $(b \leftrightarrow \bar{a})$ 的每个 B_g 。

对于用何种CNF公式替换形如 $(c \leftrightarrow (a \cdot b))$ 的公式 B_g 留给读者作为练习。用这种方法, 对每个 B_g 进行替换所得的CNF公式 S' 恰好对应于电路 C 的每个输入和逻辑门。为了构造最终的布尔公式 S , 定义 $S = S' \cdot y$, 其中 y 是与门输出有关的变量, 这个门定义了 C 自身的值。因此当且仅当 S 是可满足时, C 是可满足的。此外, 由 C 构造 S 的过程建立了关于 C 的输入和逻辑门的常数规模的子表达式, 因此, 构造过程需要多项式时间。于是, 这种局部-替换归约提供了如下定理。

定理13.3 CNF-SAT问题是NP完全问题。

606

2. 3SAT

考虑3SAT问题, 它取布尔公式 S , 公式 S 是一个合取范式 (CNF), 其中的每个子句正好有3个文字, 并询问 S 是否是可满足的。如果布尔公式是由AND连接的子句集组成, 并且每个子句由OR将文字集连接而成, 则该布尔公式在CNF中。例如, 以下公式是3SAT的一个实例:

$$(\bar{x}_1 + x_2 + \bar{x}_7)(x_3 + \bar{x}_5 + x_6)(\bar{x}_2 + x_4 + \bar{x}_6)(x_1 + x_5 + \bar{x}_8)$$

因此, 3SAT问题是CNF-SAT问题的一个受限版本。这里需要注意的是, 我们不能利用NP困难度的限制形式证明, 这是因为这种证明只适合于归约一个受限的版本到更一般的形式。这里我们利用局部替换形式, 证明3SAT是NP完全问题。有趣的是, 2SAT问题 (其中的每个子句只有两个文字) 可在多项式内求解 (见习题C-13.4和C-13.5)。

注意3SAT在NP中, 因为可以构造一个非确定性的多项式时间的算法, 取CNF公式 S , 它的每个子句有3个文字, 猜测 S 的一个布尔值指派, 然后计算 S , 看其值是否为1。

为了证明3SAT是NP困难的, 我们在多项式时间内将CNF-SAT问题归约到3SAT问题。设 C 是给定的布尔公式, 它是CNF范式。对于 C 中的每个子句 C_i , 进行以下局部替换:

- 如果 $C_i = (a)$, 即它只有一项, 也可能是一个非变量, 那么用 $S_i = (a + b + c) \cdot (a + \bar{b} + \bar{c}) \cdot (a + b + \bar{c}) \cdot (a + \bar{b} + c)$ 替换 C_i , 其中 b 和 c 是未在其他地方使用的新变量。

- 如果 $C_i = (a + b)$ ，即它有两项，那么用 $S_i = (a + b + c) \cdot (a + b + \bar{c})$ 替换 C_i ，其中 c 是未在其他地方使用的新变量。
- 如果 $C_i = (a + b + c)$ ，即它有三项，那么设 $S_i = C_i$ 。
- 如果 $C_i = (a_1 + a_2 + a_3 + \dots + a_k)$ ，即它有 $k > 3$ 项，那么用 $S_i = (a_1 + a_2 + b_1) \cdot (\bar{b}_1 + a_3 + b_2) \cdot (\bar{b}_2 + a_4 + b_3) \dots (\bar{b}_{k-3} + a_{k-1} + b_{k-2}) \cdot (\bar{b}_{k-2} + a_k)$ 替换 C_i ，其中 b_1, b_2, \dots, b_{k-1} 是未在其他地方使用的新变量。

注意对新引入变量的赋值是完全独立的。不论赋给它们何值，当且仅当小公式 S_i 的值为1时，子句 C_i 的值也为1。因此，当且仅当 S 求值为1时，原子句 C 求值为1。此外，每个子句的规模至多增加常数因子，且所涉及的计算是简单替换。因此，这就证明了在多项式时间内如何将CNF-SAT问题的实例归约成为等价的3SAT问题。结合这一点以及早先证明3SAT是在NP中，得出以下定理。

607

定理13.4 3SAT问题是NP完全问题。

13.3.2 VERTEX-COVER

回忆引理13.4中VERTEX-COVER问题取图 G 和整数 k ，询问是否存在 G 的至多包含 k 个顶点的顶点覆盖。形式上，VERTEX-COVER询问是否存在大小至多为 k 的顶点子集 C ，满足对于每条边 (v, w) ，有 $v \in C$ 或 $w \in C$ 。引理13.4中已经证明，VERTEX-COVER是在NP中。以下例子促进了对这个问题的研究。

示例13.1 假定图 G 表示计算机网络，图中顶点表示路由器，边表示物理连接。进一步假定希望用新的但是昂贵的特殊路由器升级网络中的某些路由器，这些新路由器可以对易于发生的连接进行复杂监控操作。我们想要确定 k 个新路由器是否足够用于监控网络中的每条连接，这个问题就是顶点覆盖问题的应用实例。

通过将3SAT问题在多项式时间内归约到顶点覆盖问题，证明该问题是NP困难的。这个归约过程在两个方面非常令人感兴趣。一方面，它表明可将一个逻辑问题归约到图问题，另一方面，它说明了分量设计证明技术的应用。

设 S 是给定的3SAT问题的实例，即它是每个子句正好有3个文字的一个CNF范式。构造图 G 和整数 k ，使得 G 的顶点覆盖大小至多为 k ，当且仅当 S 是可满足的。通过添加下述各项来开始进行构造：

- 对于公式 S 中所用的每个变量 x_i ，向图 G 中添加两个顶点，一个标记为 x_i ，另一个标记为 \bar{x}_i ，同时向 G 中添加一条边 (x_i, \bar{x}_i) （注意：这些标记为了方便，在 G 的构造完成之后，如果它是VERTEX-COVER看上去的一个实例，可用整数对图中顶点重新编号）。

每条边 (x_i, \bar{x}_i) 是一个“真值设置”分量，因为对于在 G 中的这条边，顶点覆盖必定包括 x_i 或 \bar{x}_i 中至少一个顶点。此外，添加如下项：

- 对于 S 中的每个子句 $C_i = (a + b + c)$ ，构造一个由顶点 i_1, i_2 和 i_3 ，以及边 (i_1, i_2) 、 (i_2, i_3) 和 (i_3, i_1) 组成的三角形。

注意，任何顶点覆盖一定至少包括集合 $\{i_1, i_2, i_3\}$ 中的两个顶点。每个这样的三角形是一个“强制满足”分量。然后对于每个子句 $C_i = (a + b + c)$ ，通过添加边 (i_1, a) 、 (i_2, b) 和 (i_3, c) ，将这两种类型的分量连接起来（如图13-5所示）。最终，我们设置整型参数 $k = n + 2m$ ，其中 n 是 S 中的变量数， m 是子句数。因此，如果存在至多为 k 的顶点覆盖，这个覆盖的大小一定恰好等于 k ，这就完成了VERTEX-COVER问题实例的构造。

608

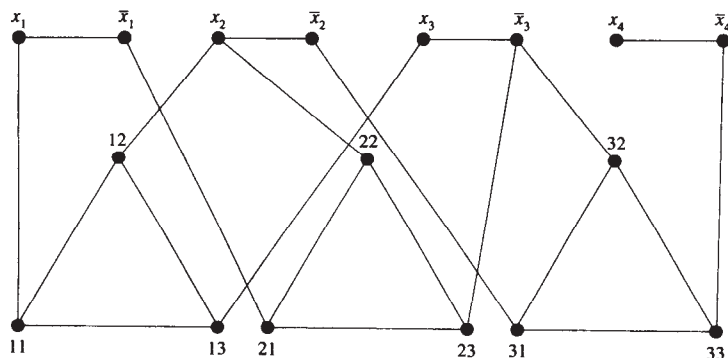


图13-5 由公式 $S = (x_1 + x_2 + x_3) \cdot (\bar{x}_1 + x_2 + \bar{x}_3) \cdot (\bar{x}_2 + \bar{x}_3 + x_4)$ 构造的VERTEX-COVER问题的实例的示例图 G

这个构造显然在多项式时间内运行，因此我们考察它的正确性。假定 S 中存在变量的一个布尔值指派值，使得 S 是可满足的。由 S 构造的图 G 可得，通过可满足性赋值，可以构造顶点 C 的子集，包含满足指派赋值为1的每个文字 a （在真值设置的分量中）。同样，对于每个子句 $C_i = (a + b + c)$ ，满足指派使得 a 、 b 或 c 中至少有一个为1。不论是哪一个为1（如果存在tie，则任取其一），将另外两个包含在子集 C 中。集合 C 的大小为 $n + 2m$ 。此外，注意在真值设置分量以及子句满足分量中的每条边都会被覆盖。依附于子句满足分量的每三条边中的两条也会被覆盖。此外，需要注意的是，对于依附于关联子句 C_i 的分量的一条边，如果没有被这个分量中的顶点覆盖，则一定会被 C 中标以文字的结点所覆盖， C_i 中对应的文字为1。

反过来，假定存在大小至多为 $n + 2m$ 的顶点覆盖 C 。由构造可知，这个集合的大小恰好是 $n + 2m$ ，因为它一定包含每个真值设置分量中的一个顶点，包含每个子句满足分量中的两个顶点。这就剩下依附于子句满足分量中的一条边，没有被子句满足分量中的顶点所覆盖；因此这条边一定被标以文字的另一个端点所覆盖。因此，可以将 S 中与这个结点关联的文字赋值为1，且 S 中的每个子句都会被满足；因此 S 中的所有子句和文字也都会被满足。于是，当且仅当图 G 存在大小至多为 k 的顶点覆盖时， S 是可满足的，由此可得以下定理。

定理13.5 VERTEX-COVER是NP完全问题。

如前所述，上述归约过程说明了分量设计技术。在图 G 中，构造真值设置的分量和子句满足的分量，以增强子句 S 中的某些重要性质。

609

13.3.3 CLIQUE 和 SET-COVER

与VERTEX-COVER问题一样，还有一些问题也涉及从一个较大集合中选择对象的子集，目标是优化所选子集的大小，同时满足一个重要的性质。在这一节里，研究两个这样的问题，它们是CLIQUE问题和SET-COVER问题。

1. CLIQUE

图 G 中的一个团（clique）是顶点的一个子集 C ，满足对于 C 中的每个顶点 v 和 w ， $v \neq w$ ， (v, w) 是一条边，即对于 C 中每一对不同的顶点，存在一条边。CLIQUE问题取图 G 和整数 k ，询问 G 中是否存在大小至少为 k 的团。

我们将证明CLIQUE在NP中这个问题留作练习。为了证明团问题是NP困难的，我们把VERTEX-COVER问题归约到这个问题。于是，设 (G, k) 是VERTEX-COVER问题的一个实例。对于

CLIQUE问题, 构造补图 G^C , 它和图 G 有相同顶点集, 但是, 当且仅当 $(v, w) \notin G$ 时, 具有 G^C 中的边 (v, w) , 其中 $v \neq w$ 。定义CLIQUE的整型参数为 $n - k$, 其中 k 是VERTEX-COVER的整型参数。这个构造过程需要多项式时间, 主要是归约时间, 因为当且仅当图 G 中存在大小至多为 k 的顶点覆盖时, 图 G^C 中存在大小至少为 $n - k$ 的团 (如图13-6所示)。由此可得定理13.6。

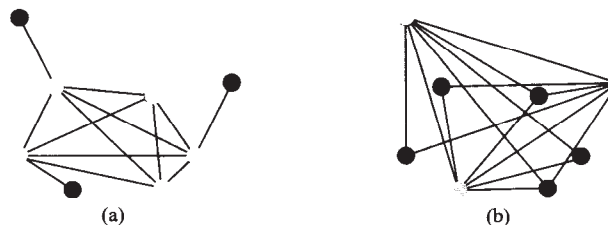


图13-6 图 G 说明了CLIQUE问题是NP困难的证明。(a)图 G 中存在大小为5的团, 结点用灰色表示出;
(b)图 G^C 中存在大小为3的顶点覆盖, 结点用灰色表示出

定理13.6 CLIQUE问题是NP完全问题。

注意, 通过局部替换, 问题的证明变得如此简单。有趣的是, 下一个归约也是基于局部替换技术, 甚至更简单。

610

2. SET-COVER

SET-COVER问题取 m 个集合 S_1, S_2, \dots, S_m 以及整数 k 作为输入, 并询问是否存在 k 个集合 $S_{i_1}, S_{i_2}, \dots, S_{i_k}$, 满足

$$\bigcup_{i=1}^m S_i = \bigcup_{j=1}^k S_{i_j}$$

即 k 个子集的并集包含原 m 个集合的并集中的每个元素。

我们将证明SET-COVER问题在NP中留作习题 (R-13.14)。至于归约, 可以由VERTEX-COVER问题的实例 G 和 k 来定义SET-COVER问题的实例, 即对于 G 中的每个顶点 v , 存在集合 S_v , 包含了 G 中依附于 v 的所有边。显然, 当且仅当 G 中存在大小为 k 的顶点覆盖时, 在这些集合 S_v 中存在一个大小为 k 的集合覆盖 (如图13-7所示)。

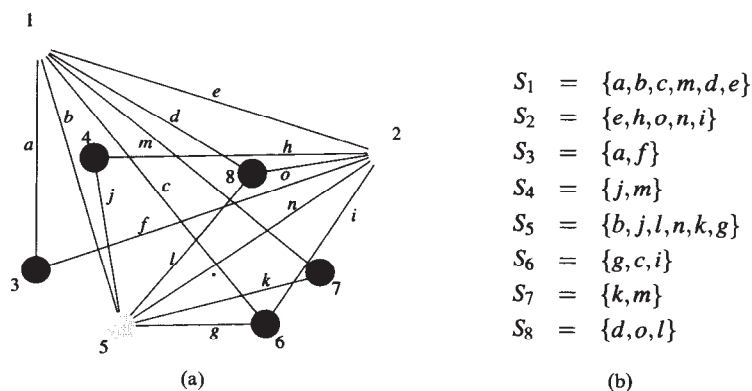


图13-7 图 G 说明了SET-COVER问题是NP困难的证明。顶点编号为1~8, 边上标以 $a \sim o$ 中的字母。
(a)显示图 G 中的顶点覆盖为3, 结点用灰色表示出; (b)显示关联 G 中每个顶点的集合, 并且集合的下标确定了关联的顶点。注意 $S_1 \cup S_2 \cup S_5$ 包含了 G 中的所有边

由此可得以下定理。

定理13.7 集合覆盖问题是NP完全问题。

这个归约过程说明了把一个图问题转换成一个集合问题有多容易。下一节说明把一个图问题归约到数问题实际上有多容易。

611

13.3.4 SUBSET-SUM 和 KNAPSACK

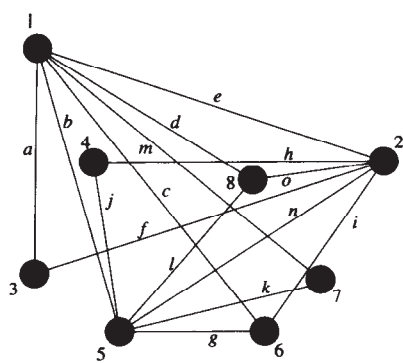
某些困难的问题只与数有关。在这些情况下,我们必须额外考虑输入的位数,因为某些数非常大。为了阐明数的大小所起的作用,研究人员称一个问题 L 是强NP困难的(strongly NP-hard),如果将其输入中的每个数限制到多项式的数量级(按位计算),它仍然是NP困难的。例如,如果大小为 n 的 x 中的每个数 i 可用 $O(\log n)$ 位表示,就属于这种情况。有趣的是我们本节研究的关于数的问题不是强NP困难的(见习题C-13.12和C-13.13)。

1. SUBSET-SUM

在SUBSET-SUM问题中,给定 n 个整数的集合 S 以及一个整数 k ,询问集合 S 中是否存在其和为 k 的整数子集。以下例子可以引出这个问题。

示例13.2 假定我们有一台Internet Web服务器,给定一个下载请求的集合。对于每个下载请求,可以容易确定请求下载文件的大小。因此,可以将每个网络下载请求简单地抽象为一个整数——请求下载文件的大小。给定整数的集合,目标是确定这些整数(请求)的一个子集,使得子集中的整数之和恰好等于服务器一分钟内可容纳的带宽。不幸的是,这个问题是子集和数问题的一个实例。而且,由于它是NP完全问题,当网络服务器带宽增加和请求处理能力提高时,这个问题变得更难求解。

SUBSET-SUM问题初看起来相当简单,证明它在NP中相当直接(见习题R-13.15)。然而它是NP完全问题。设 G 和 k 是给定VERTEX-COVER问题的实例,对 G 中的顶点从1到 n 进行编号,对边从1到 m 进行编号,构造 G 的依附矩阵(incidence matrix) H ,使得当且仅当编号为 j 的边依附于编号为 i 的顶点时 $H[i, j] = 1$; 否则, $H[i, j] = 0$ (如图13-8所示)。



(a)

H	1	2	3	4	5	6	7	8
a	1	0	1	0	0	0	0	0
b	1	0	0	0	1	0	0	0
c	1	0	0	0	0	1	0	0
d	1	0	0	0	0	0	0	1
e	1	1	0	0	0	0	0	0
f	0	1	1	0	0	0	0	0
g	0	0	0	0	1	1	0	0
h	0	1	0	1	0	0	0	0
i	0	1	0	0	0	1	0	0
j	0	0	0	1	1	0	0	0
k	0	0	0	0	1	0	1	0
l	0	0	0	0	1	0	0	1
m	0	1	0	1	0	0	0	0
n	0	1	0	0	1	0	0	0
o	0	1	0	0	0	0	0	1

(b)

图13-8 图 G 说明了SUBSET-SUM问题是NP困难的证明。顶点从1到8编号,边上标以 $a \sim o$ 中的字母。
(a)显示图 G ; (b)显示 G 的依附矩阵 H 。注意对于顶点1、2和5,每条边在一列或多列中都有一个1

利用 H 定义某些公认的大数(仍然为多项式大小), 作为SUBSET-SUM问题的输入, 即对于 H 中每一行 i , 它对依附于顶点 i 的所有边进行编码, 构造数

$$a_i = 4^{m+1} + \sum_{j=1}^m H[i, j]4^j$$

注意这个求和式将第 i 行的每个为1的元素按4的不同幂相加, 然后再加上一个4的较大次幂, 以便作为一种好的测度。对于 a_i 中的每个4的幂, 除了最大的一个 a_i 之外, a_i 中的值集定义了归约的一个“依附分量”, 都对对应顶点 i 和某些边的一种可能的依附关系。

612

除了上述的依附分量, 还定义“边覆盖”分量, 其中对于每条边 j , 定义数

$$b_j = 4^j$$

用这些数的子集设置我们希望得到的和

$$k' = k4^{m+1} + \sum_{j=1}^m 2 \cdot 4^j$$

其中 k 为VERTEX-COVER实例的整型参数。

接着考虑如何用多项式时间进行归约。假定图 G 存在大小为 k 的顶点覆盖 $C = \{i_1, i_2, \dots, i_k\}$, 通过取下标在 C 中的那些 a_i , 即 $r = 1, 2, \dots, k$ 的那些 a_{i_r} , 将它们添加到 k' 中, 构造值的集合。此外, 对于 G 中编号为 j 的每条边, 如果 j 只有一个端点包含在 C 中, 则也将 b_j 包含在子集中。这组数之和为 k' , 因为它包括 4^{m+1} 的 k 个值, 加上每个 4^j 的两个值(如果这条边 j 有两个端点在 C 中, 则这两个值为两个 a_{i_r} 值; 如果 C 只包含边 j 的一个端点, 则这两个值一个为 a_{i_r} , 另一个为 b_j)。

假定存在和为 k' 的数的子集。因为 k' 包含 4^{m+1} 的 k 个值, 它必定恰好包含 k 个 a_i 的值。对于每个这样的 a_i , 将顶点 i 包含在覆盖中。这样的集合是一个覆盖, 因为对于与 4^j 的幂对应的每条边 j , 必定为这个和贡献两个值, 由于只有一个值来自 b_j , 则另一个值至少必定来自所选 a_i 中的一个值。由此可得:

613

定理13.8 SUBSET-SUM是NP完全问题。

2. KNAPSACK

在图13-9所示的KNAPSACK(背包)问题中, 给定编号为 $1 \sim n$ 的物品集合 S 。每件物品 i 大小为整数 s_i , 价值为 w_i 。还给定两个整型参数 s 和 w , 并询问是否存在 S 的子集 T , 满足

$$\sum_{i \in T} s_i \leq s \text{ 且 } \sum_{i \in T} w_i \geq w$$

上面定义的KNAPSACK问题是5.3.3节讨论的“0-1背包”优化问题的判定版本。

以下因特网应用促进了KNAPSACK问题的研究。

示例13.3 假定我们在因特网拍卖网站上销售 s 件物品。预期的买主 i 希望以总价 w_i 美元用多次抽签投标方式购买 s_i 件物品。这样, 多次抽签请求不能分解物品(即买主只想要 s_i 件物品), 那么确定是否能从这次拍卖中挣得 w 美元就引出了KNAPSACK问题(如果能够分解, 则拍卖优化问题会产生分数KNAPSACK问题, 可以利用贪心算法有效解决这个问题)。

KNAPSACK问题是在NP中, 因为可以构造一个猜测放在子集 T 中的物品的非确定性多项式时间的算法, 然后分别验证它们是否满足关于 s 和 w 的约束条件。

KNAPSACK问题也是NP困难的,实际上SUBSET-SUM问题是它的特例。特别是给定SUBSET-SUM问题的任何数的实例,都存在对应的KNAPSACK问题的实例,设置每个 $w_i = s_i$ 为SUBSET-SUM实例中的一个值,目标是大小 s 和价值 w 都等于 k ,其中 k 是表示SUBSET-SUM问题中的和数。因此,利用限制证明技术,可得以下定理。

定理13.9 KNAPSACK问题是NP完全问题。

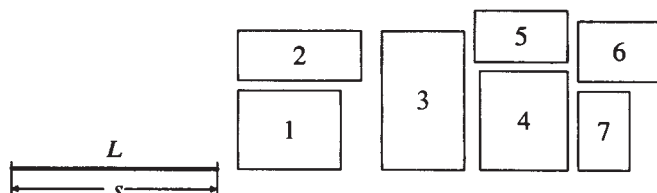


图13-9 从几何角度看KNAPSACK问题。给定一条长为 s 的直线 L , n 个矩形集合,可以把矩形的一个子集转换为其底边在 L 上,使得接触 L 的矩形的整个区域面积至少为 w 。因此,矩形 i 的宽度为 s_i ,对应面积为 w_i

614

13.3.5 HAMILTONIAN-CYCLE 和 TSP

最后讨论的两个NP完全问题是查找图中是否存在某种回路的问题。例如,这样的问题在机器人和绘图打印机的行程优化中具有广泛的应用。

1. 哈密尔顿回路问题

回忆引理13.2, HAMILTONIAN-CYCLE (哈密尔顿回路)问题是指:取图 G ,并询问 G 中是否存在简单回路,它将访问 G 中的每个顶点刚好一次,并回到它的起始顶点(如图13-10a所示)。再次回忆引理13.2可知, HAMILTONIAN-CYCLE问题在NP中,为了证明这个问题是NP完全问题,我们利用归约的分量设计,把VERTEX-COVER问题归约到这个问题。

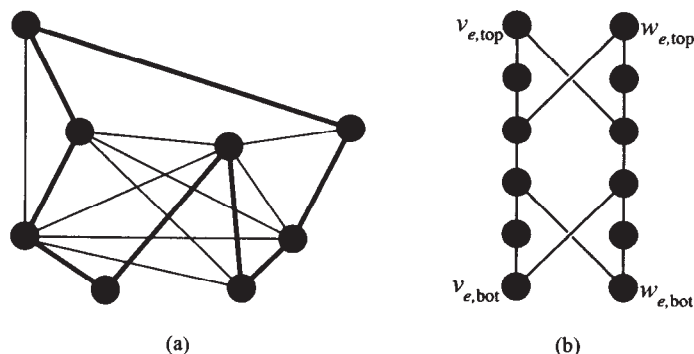


图13-10 证明HAMILTONIAN-CYCLE问题是NP完全问题的图示说明。(a)给出一个哈密尔顿回路(粗体显示)的示例图;(b)利用一个强制覆盖子图 H_e 说明HAMILTONIAN-CYCLE问题是NP困难的

设 G 和 k 是给定VERTEX-COVER问题的实例。构造一个图 H ,当且仅当 G 存在大小为 k 的顶点覆盖时, H 含有一个哈密尔顿回路。首先,构造 k 个初始不相连的顶点集合 $X = \{x_1, x_2, \dots, x_k\}$ 作为 H 。这个顶点集合作为“覆盖选择”分量,因为它们可以确定 G 中哪些结点应该包含在顶点覆盖中。此外,对于 G 中的每条边 $e = (v, w)$,在 H 中构造一个“强制覆盖”子图 H_e ,这个子图 H_e 有12个顶点和14条边,如图13-10b所示。

对于边 $e = (v, w)$, 在强制覆盖 H_e 中, 有6个顶点与 v 对应, 另外6个顶点与 w 对应。另外, 在强制覆盖 H_e 中, 将与顶点 v 对应的两个顶点标记为 $v_{e, \text{top}}$ 和 $v_{e, \text{bot}}$, 将与顶点 w 对应的两个顶点标记为 $w_{e, \text{top}}$ 和 $w_{e, \text{bot}}$ 。 H_e 中只有这些顶点连向 H_e 外的 H 中的其他任何顶点。

615

因此, 哈密尔顿回路只能按照图13-11所示的三种方式之一访问 H_e 中的结点。

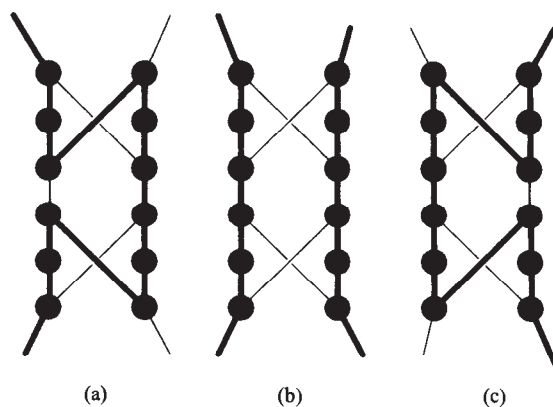


图13-11 在强制覆盖 H_e 中, 哈密尔顿回路访问边的三种可能的方式

按照两种方式, 将每个强制覆盖 H_e 中的重要顶点与 H 中的其他顶点连接。一种对应覆盖选择分量, 另一种对应强制覆盖分量。对于覆盖选择分量, 在 X 中从每个顶点到每个顶点 $v_{e, \text{top}}$ 和 $v_{e, \text{bot}}$ 之间添加一条边, 即向 H 中添加 $2kn$ 条边, 其中 n 是 G 中的顶点数。

对于强制覆盖分量, 依次考虑 G 中的每个顶点 v , 对于每个这样的顶点 v , 设 $\{e_1, e_2, \dots, e_{d(v)}\}$ 是 G 中依附于 v 的边的列表。利用这个列表, 通过连接 H_{e_i} 中的顶点 $v_{e_i, \text{bot}}$ 与 $H_{e_{i+1}}$ 中的顶点 $v_{e_{i+1}, \text{top}}$, 创建 H 中的边, 其中 $i = 1, 2, \dots, d-1$ (如图13-12所示)。称用这种方式连接的 H_{e_i} 分量为 v 的覆盖线索 (covering thread), 这就完成了图 H 的构造。注意, 这个计算的运行时间为 G 的大小的多项式时间。

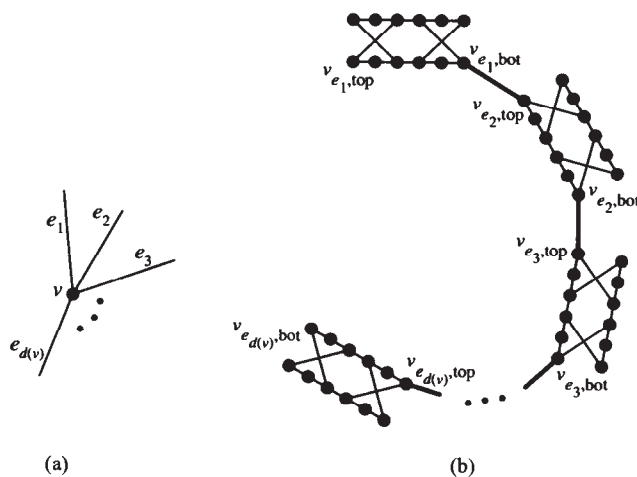


图13-12 连接强制覆盖: (a) G 中的顶点 v 和它所依附的边集合 $\{e_1, e_2, \dots, e_{d(v)}\}$; (b) 对于依附于顶点 v 的边, 在 H 中 H_{e_i} 之间创建的连接

声称当且仅当 H 中存在哈密尔顿回路时, G 中存在大小为 k 的顶点覆盖。首先假定 G 中存在大

小为 k 的顶点覆盖, 设 $C = \{v_{i_1}, v_{i_2}, \dots, v_{i_k}\}$ 是这样一个覆盖。通过连接一系列的路径 P_j , 构造 H 中的一个哈密顿回路, 每条 P_j 在 x_j 处开始, 在 x_{j+1} 处结束, 其中 $j = 1, 2, \dots, k-1$, 只不过最后一条路径 P_k 在 x_k 处开始, 在 x_1 处结束。按如下方式构造这条路径 P_j 。从 x_j 开始, 然后访问 H 中 v_{i_j} 的整个覆盖线索, 返回到 x_{j+1} (或 x_1 , 如果 $j = k$)。对于 v_{i_j} 的覆盖线索中的每个强制覆盖子图 H_e , 都会在这条路径 P_j 上被访问, 不失一般性, 记 e 为 (v_{i_j}, w) 。如果 w 也不在 C 中, 那么按照图13-11a或图13-11c访问这个 H_e (关于 v_{i_j})。如果 w 也在 C 中, 那么按照图13-11b访问这个 H_e 。用这种方法, 我们访问 H 中的每个顶点恰好一次, 因为 C 是 G 的一个顶点覆盖。因此, 我们所构造的这个回路实际上是哈密顿回路。

相反, 如果假定 H 具有一个哈密顿回路。因为这个回路一定访问了 X 中的所有顶点, 将这个回路分成 k 条路径 P_1, P_2, \dots, P_k , 每条路径都开始和结束于 X 中的一个顶点。由强制覆盖子图 H_e 的结构以及连接它们的方式可知, 每个 P_j 必定遍历 G 中顶点 v 的覆盖线索的一部分(也可能是全部)。设 C 是 G 中所有这样的顶点集。因为哈密顿回路必定包括每个强制覆盖 H_e 中的顶点, 并且每个这样的子图必定会以对应于 e 的一个(或两个)端点的方式被遍历, 因此 C 必定是 G 的一个顶点覆盖。

因此, 当且仅当 H 中存在哈密顿回路时, G 中存在大小为 k 的覆盖。由此可得以下定理。

定理13.10 HAMILTONIAN-CYCLE问题是NP完全问题。

2. TSP

在旅行推销员问题(traveling salesman problem, TSP)中, 给定一个整型参数 k 和图 G , 满足 G 中的每条边 e 被赋予一个整数代价 $c(e)$, 并询问 G 中是否存在一个回路, 它将访问 G 中的所有顶点(可能多次访问), 且总代价至多为 k 。证明TSP在NP中也是很容易的, 只需猜测顶点的一个序列, 然后验证它是否形成 G 中代价至多为 k 的回路。因为HAMILTONIAN-CYCLE问题是TSP问题的特例, 因而证明TSP问题是NP完全问题也是容易的。也就是说, 给定HAMILTONIAN-CYCLE问题的一个实例 G , 通过给 G 中的边赋予代价 $c(e)=1$, 并设置整型参数 $k=n$, 可以构造TSP问题的一个实例, 其中 n 是图 G 中的顶点数。因此, 利用归约的限制形式, 得到以下定理。

定理13.11 TSP问题是NP完全问题。

13.4 近似算法

近似算法(approximation algorithm)是处理优化问题的NP完全性的一种方法。这样的算法通常比问题的精确算法解更有效。但不能保证找到问题的最优解。在这一节里, 研究构造和分析困难的优化问题的近似算法的一些方法。

在一般情况下, 我们有某个问题的实例 x , 它可能是上面讨论的一组数、一个图等的编码。此外, 在对求解 x 感兴趣的问题中, 常常有许多 x 的可行(feasible)解, 这样的可行解集合定义为 \mathcal{F} 。

对于任何解 $S \in \mathcal{F}$, 还有一个代价函数 c 用来确定一个数值代价 $c(S)$ 。在一般优化问题中, 我们对于找出 \mathcal{F} 中满足

$$c(S) = OPT = \min\{c(T) : T \in \mathcal{F}\}$$

的解 S 感兴趣, 即希望找出最小代价的解。也可以把优化问题形式化为找出最大值的问题, 只要简单地把上述相关“min”换成“max”即可。为使本节的讨论简单, 除非特别说明, 我们的优

616

617

化目标是最小化。

近似算法的目标是在一个合理的时间内尽可能地向最优值靠近。正如在这一章里所做的那样，我们认为这个合理的时间至多为多项式时间。

理想情况下，希望给出近似算法与最优值 OPT 接近程度的保证。对于最小化的优化问题，称该问题的 δ -近似算法（ δ -approximation）是指算法返回的可行解 S （即 $S \in \mathcal{F}$ ）满足

$$c(S) \leq \delta OPT$$

对于最大化问题， δ -近似算法会保证 $OPT \leq \delta c(S)$ 。或者，一般而言，有

$$\delta \leq \max \{c(S)/OPT, OPT/c(S)\}$$

在本节其余部分，研究对于不同的 δ 值，构造 δ -近似算法的问题。从理想情况开始，关注近似因子近似的程度。

618

13.4.1 多项式时间的近似模式

可以构造某些问题的 δ -近似算法，它的运行时间为多项式时间，并且 $\delta = 1 + \varepsilon$ ，其中 $\varepsilon > 0$ 为固定值。这样一个算法集合的运行时间依赖于输入大小 n 以及固定值 ε 。称这样的算法集合为多项式时间的近似模式（polynomial-time approximation scheme, PTAS）。当得到给定优化问题的多项式时间的近似模式时，可以基于能够花得起的时间调整性能保证。理想情况下，这个运行时间介于 $n \sim 1/\varepsilon$ 之间。在这种情况下，称之为完全多项式时间的近似模式（fully polynomial-time approximation scheme）。

多项式时间的近似模式利用某些困难问题具有的一个性质，即它们是可伸缩的。如果问题的一个实例 x 可以通过伸缩函数 c 转换成等价的实例 x' （即具有相同优化解的实例），则称这个问题是可伸缩的（rescalable）。例如，TSP是可伸缩的。给定TSP的一个实例 G ，在它的每对顶点之间的距离上乘以伸缩因子 s ，可以构造等价的实例 G' 。 G' 的TSP路径和 G 的路径相同，尽管它现在的代价是原来的 s 倍。

1. KNAPSACK问题的完全多项式时间的近似模式

为了更具体地加以说明，我们给出一个众所周知的优化问题即KNAPSACK问题（5.1.1节和13.3.4节）的完全多项式的近似模式。在这个问题的优化版本中，给定标号从1到 n 物品集合 S ，以及大小约束 s 。 S 中的每件物品 i 有整数大小 s_i 和价值 w_i ，要求找出 S 的一个子集 T ，满足 T 最大化价值

$$w = \sum_{i \in T} w_i, \text{ 同时满足 } \sum_{i \in T} s_i \leq s$$

我们希望PTAS产生 $(1+\varepsilon)$ 近似，其中 ε 为固定常数，即这样一个算法应该找到满足大小约束的子集 T' ，使得如果定义 $w' = \sum_{i \in T'} w_i$ ，那么

$$OPT \leq (1+\varepsilon)w'$$

其中 OPT 是满足总大小约束的所有可能子集的最优价值和。为证明这个不等式成立，对于 $0 < \varepsilon < 1$ ，实际上证明

$$w' \geq (1-\varepsilon/2)OPT$$

然而这是一个充分条件，因为对于任意固定的 $0 < \varepsilon < 1$,

619

$$\frac{1}{1-\varepsilon/2} < 1+\varepsilon$$

为了导出KNAPSACK问题的PTAS，我们利用这个问题是可伸缩的这一事实。假设给定 ε 值，

且 $0 < \varepsilon < 1$ 。设 w_{\max} 表示 S 中物品的最大价值。不失一般性, 假设每件物品的大小至多为 s (因为大于此数的物品不能装入背包中)。因此, 价值 w_{\max} 的物品定义了优化值的一个下界, 即 $w_{\max} \leq OPT$ 。同样, 可以定义最优解的一个上界, 这就是背包至多能包含 S 中的所有 n 件物品, 每件物品的价值至多为 w_{\max} 。因此, $OPT \leq nw_{\max}$ 。利用 KNAPSACK 问题的可伸缩性, 把每个价值 w 向下取整得到 w' , 即最近的更小倍数 M 为 $\varepsilon w_{\max}/2n$ 。取整后的 S 表示为 S' , 利用 OPT' 表示取整后 S' 的解。注意, 通过简单代换得到 $OPT \leq 2n^2 M/\varepsilon$ 。此外, $OPT' \leq OPT$, 因为对 S 中的每个值向下取整形成 S' , 因此, $OPT' \leq 2n^2 M/\varepsilon$ 。

于是, 现在转到 S 的 KNAPSACK 问题的向下取整后 S' 的解。因为 S' 中的每个值的值是 M 的倍数, 从 S' 中所取物品的可达价值也是 M 的倍数。而且, 由于 OPT' 的上界, 只需考虑 $N = \lceil 2n^2/\varepsilon \rceil$ 个这样的倍数。可以利用动态规划 (5.3 节) 构造一个用于找出 S' 最优价值的有效算法。尤其是, 定义参数

$s[i, j] = \{1, 2, \dots, j\}$ 中价值 iM 的最小物品集大小

设计用于求解取整 KNAPSACK 问题的动态规划算法的关键是

$$s[i, j] = \min \{s[i, j-1], s_j + s[i - (w'_j/M), j-1]\}$$

其中 $i = 1, 2, \dots, N$, $j = 1, 2, \dots, n$ (见图 13-13)。

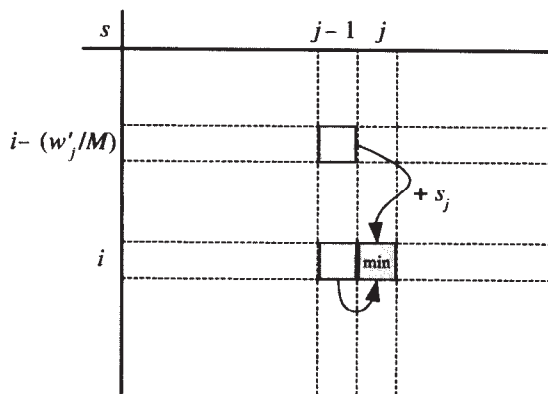


图 13-13 KNAPSACK 问题的可伸缩版本的动态规划中所用 $s[i, j]$ 方程的说明

620

上述 $s[i, j]$ 方程由物品 j 是否对达到价值 iM 的最小方式有所贡献的事实而得, 其中物品在 $\{1, 2, \dots, j\}$ 中。此外, 注意对于 $j = 0$ 的情形, 当背包中没有物品时, 那么

$$s[i, 0] = +\infty$$

其中 $i = 1, 2, \dots, N$, 即这个大小是未定义的。此外,

$$s[0, j] = 0$$

其中 $j = 1, 2, \dots, n$, 因为背包中没有物品, 价值总是为 0。定义最优值为

$$OPT' = \max \{iM: s[i, n] \leq s\}$$

这就是 PTAS 算法输出的值。

2. KNAPSACK 问题的 PTAS 分析

可以容易地把上述描述转换成一个动态规划算法, 它计算 OPT' 的时间为 $O(n^3/\varepsilon)$ 。这个算法给出一个最优解的值, 但是可以容易地把这个计算大小的动态规划算法转换成为计算实际物品集合的算法。

考虑 OPT' 与 OPT 相近的程度。回忆我们把每个物品 i 的 w_i 价值减少到至多为 $M = \varepsilon w_{\max}/2n$ 。因此,

$$OPT' \geq OPT - \varepsilon w_{\max}/2,$$

因为最优解至多包含 n 件物品。由于 $OPT \geq w_{\max}$, 这反过来蕴涵着

$$OPT' \geq OPT - \varepsilon OPT/2 = (1 - \varepsilon/2)OPT.$$

因此, $OPT \leq (1 + \varepsilon)OPT'$, 这正是我们想要证明的。我们的近似算法的运行时间为 $O(n^3/\varepsilon)$ 。对于 $\varepsilon > 0$, 设计有效算法的模式引出一种完全多项式的近似模式, 因为这个运行时间是介于 n 和 $1/\varepsilon$ 之间的多项式。由这个事实可得以下结论。

621

定理13.12 背包优化问题有运行时间为 $O(n^3/\varepsilon)$ 的完全多项式近似模式, 达到 $(1 + \varepsilon)$ 近似因子。其中 n 是KNAPSACK问题实例中的物品数, $0 < \varepsilon < 1$ 是给定的固定常数。

13.4.2 VERTEX-COVER 的 2-近似算法

并不总是能够为困难的问题设计多项式时间的近似模式, 更不用说完全多项式时间的近似模式了。在这种情况下, 仍然希望一个良好的近似, 但必须解决近似因子不是任意接近1的情形, 如在上一小节中讨论的那样。在这一小节里, 描述一个如何达到众所周知的NP完全问题即VERTEX-COVER问题(13.3.2节)的2-近似算法。在这个问题的优化版本中, 给定图 G , 要求产生 G 的顶点覆盖的最小集合 C , 满足 G 中的每条边依附于 C 中的某个顶点。

我们的近似算法基于贪心法, 相当简单。取图中的一条边, 把它的两个端点添加到覆盖中, 然后, 从图中删除这条边和依附它的边。算法重复这个过程, 直到没有边剩下。算法13-1中给出了这种方法的细节。

算法13-1 VERTEX-COVER的2-近似算法

```

算法 VertexCoverApprox( $G$ ):
    输入: 图 $G$ 
    输出:  $G$ 的小顶点覆盖 $C$ 
     $C \leftarrow \emptyset$ 
    while  $G$ 中仍然存在边 do
        选择 $G$ 中的一条边 $e = (v, w)$ 
        把顶点 $v$ 和 $w$ 添加到 $C$ 中
        for 每条依附于 $v$ 或 $w$ 的边 $f$  do
            从 $G$ 中删除 $f$ 
    return  $C$ 
  
```

我们把用 $O(n + m)$ 时间实现这个算法的细节留作一个简单的习题(R-13.18)。然后考虑为什么这个算法是一个2-近似算法。首先, 观察算法选择用于把 v 和 w 添加到 C 中的每条边 $e = (v, w)$ 必定包含在任意顶点覆盖中, 即 G 的任意顶点覆盖必定包含 v 或 w (可能两者都包含)。在这种情况下, 近似算法把 v 和 w 都添加到 C 中。当近似算法运行结束后, G 中没有未包含的边剩下, 因为当把顶点 v 和 w 添加到 C 中时, 已经删除了它们覆盖的所有边。因此, C 形成 G 的一个顶点覆盖。而且, C 的大小至多为 G 的最优顶点覆盖的两倍, 因为对于添加到 C 中的每两个顶点, 其中一个必定属于最优覆盖。于是, 可得以下定理。

定理13.13 给定具有 n 个顶点、 m 条边的图，VERTEX-COVER的优化版本有运行时间为 $O(n+m)$ 的2-近似算法。

622

13.4.3 TSP 特例的 2-近似算法

在旅行推销员或TSP问题的优化版本中，给定一个加权图 G ，满足 G 中的每条边 e 有整数权值 $c(e)$ ，要求找出 G 中具有最小权值的回路，且访问 G 中的所有顶点。在这一节里，我们描述一个TSP优化问题特例的2-近似算法。

1. 三角不等式

考虑TSP的一个实例，使得边上权值满足三角不等式 (triangle inequality)，即对于 G 中的任意三条边 (u, v) 、 (v, w) 和 (u, w) ，有

$$c((u, v)) + c((v, w)) \geq c((u, w))$$

此外，假定 G 中的每对顶点通过一条边相连，即 G 是一个完全图。这些性质对于任意距离测度成立。这蕴涵着 G 的最优路径将访问每个顶点恰好一次；因此，只考虑哈密尔顿回路作为TSP问题的可能解。

2. 近似算法

我们的近似算法利用了 G 的上述性质，设计一个很简单的TSP问题的近似算法，它只有三步。在第一步中，构造 G 的一棵最小生成树 M (7.3节)。在第二步中，构造 M 的欧拉路径遍历 E ，即 M 的一个遍历开始和结束于相同的点，并在每个方向上只访问 M 中的边一次 (2.3.3节)。在第三步中，在 E 的边上行进，构造路径 T ，但每当遇见 E 中两条边 (u, v) 和 (v, w) ，且满足 v 已经被访问过时，则用边 (u, w) 代替这两条边，并继续这个过程。本质上，对 M 进行前序遍历构造 T 。这个三步骤算法显然运行多项式时间 (见图13-14)。

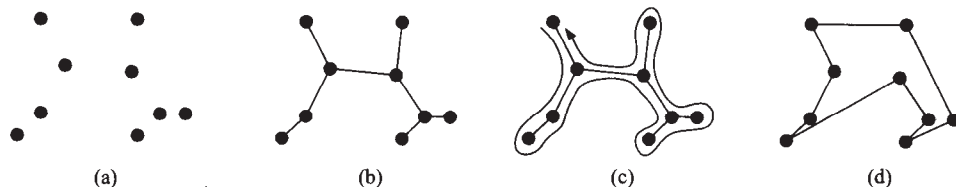


图13-14 对于满足三角不等式的图的TSP近似算法的运行例子：(a)平面上的点集；用欧几里得距离定义边上的代价（未显示出）；(b) M 的最小生成树；(c) M 的欧拉路径 E ；(d)近似的TSP路径 T

623

3. TSP近似算法分析

对于这个算法能够达到近似因子2的原因所进行的分析也很简单。对表示做一些扩展，使 $c(H)$ 表示 G 的子图 H 的边的总权值。设 T 是图 G 的最优路径。如果从 T 中删除任意一条边，得到一条路径，它当然也是一棵生成树。因此，

$$c(M) \leq c(T)$$

可以容易地把 E 的代价与 M 的代价联系起来，

$$c(E) = 2c(M)$$

因为欧拉路径 E 在每个方向上访问 M 中的边一次。最后由三角不等式可知，当构造路径 T 时，每次用边 (u, w) 代替两条边 (u, v) 和 (v, w) 并不会增加路径的代价，即

$$c(T) \leq c(E)$$

于是, 有

$$c(T) \leq 2c(T')$$

(见图13-15)。



图13-15 TSP优化问题的基于MST算法是一个2-近似算法的说明

我们把讨论概括为如下定理。

定理13.14 如果加权图是完全图, 且边上权值满足三角不等式, 那么存在图 G 的TSP优化问题的2-近似算法, 并且其运行时间为多项式时间。

这个定理主要依赖于图 G 的代价函数满足三角不等式这一事实。事实上, 如果没有这个假设, 那么TSP优化版本不存在运行时间为多项式时间的常数因子的近似算法, 除非 $P=NP$ (见习题C-13.14)。

624

13.4.4 SET-COVER 的对数近似算法

在某些情况下, 甚至难以得到运行时间为多项式时间的常数因子的近似算法。在这一节里, 研究其中一个最著名的问题, 即SET-COVER问题 (13.3.3节)。在这个问题的优化版本中, 给定集合 S_1, S_2, \dots, S_m , 它的并集是大小为 n 的全覆盖 U , 要求找出最小整数 k , 满足存在 k 个集合 $S_{i_1}, S_{i_2}, \dots, S_{i_k}$, 使

$$U = \bigcup_{i=1}^m S_i = \bigcup_{j=1}^k S_{i_j}$$

成立。尽管难以找到这个问题的多项式时间的常数因子的近似算法, 但是我们可以设计近似因子为 $O(\log n)$ 的有效算法。和其他几个针对困难问题的近似算法一样, 这个算法也基于贪心法 (5.1节)。

1. 贪心法

算法一次选择一个集合 S_{i_j} , 每次选择有最多未覆盖的集合。当 U 中的每个元素都被覆盖时, 则完成执行。算法13-2给出了一种简单伪代码描述。

算法13-2 SET-COVER的近似算法

算法 SetCoverApprox(S):

输入: 集合 S_1, S_2, \dots, S_m 的集 S , 这些集合的并集为 U

输出: S 的一个小集合覆盖

$C \leftarrow \emptyset$ {正在构建的集合覆盖}

$E \leftarrow \emptyset$ { U 中覆盖过的元素集合}

while $E \neq U$ **do**

 选择集合 S_i , 使未覆盖的元素个数最多

```

把 $S_i$ 添加到 $C$ 中
 $E \leftarrow E \cup S_i$ 
return  $C$ 

```

该算法的运行时间为多项式时间（见习题R-13.19）。

2. 贪心SET-COVER算法分析

为了分析上述SET-COVER算法的近似因子，基于支付模式（1.5节）的平摊分析进行论述，即每当近似算法选择一个集合 S_i 时，就为 S_i 中的元素支付选择的费用。

625

确切地讲，考虑算法把集合 S_i 添加到 C 中的情形，设 k 是此前 S_i 中未覆盖的元素个数。把这个集合添加到 C 中，总费用为1。因而，为此前每个未覆盖的 S_i 中的元素所支付的费用为

$$c(i) = 1/k$$

因此，覆盖的总大小等于算法支付的总费用，即

$$|C| = \sum_{i \in U} c(i)$$

为了证明一个近似界限，考虑对属于最优覆盖 C' 中的每个子集 S_j 所支付的费用。假定 S_j 属于 C' 。记作 $S_j = \{x_1, x_2, \dots, x_{n_j}\}$ ，使得 S_j 中的元素按照被算法覆盖的次序列出（任意中断tie）。现在，考虑 x_1 首先被覆盖的迭代。此时，还未选择 S_j ；因此，不论选择哪个集合，必定至少有 n_j 个未覆盖的元素。因此，为 x_1 所支付的费用至多是 $1/n_j$ 。那么，然后考虑选择 S_j 中的元素 x_l 。在最坏情况下，还未选择 S_j （算法可能永不选择这个 S_j ）。在这次迭代中，无论选择哪个集合，最坏情况下至少有 $n_j - l + 1$ 个未覆盖的元素；因此， x_l 至多被支付 $1/(n_j - l + 1)$ 的费用。于是，对 S_j 中元素支付的总费用至多为

$$\sum_{l=1}^{n_j} \frac{1}{n_j - l + 1} = \sum_{l=1}^{n_j} \frac{1}{l}$$

这是熟悉的调和数 H_{n_j} 。已知（例如，参见附录） H_{n_j} 是 $O(\log n_j)$ 。设 $c(S_j)$ 表示对属于最优覆盖 C' 的每个子集 S_j 的所有元素所支付的总费用。我们的支付模式蕴涵着 $c(S_j)$ 是 $O(\log n_j)$ 。因此，在集合 C' 上求和，可得

$$\begin{aligned} \sum_{S_j \in C'} c(S_j) &\leq \sum_{S_j \in C'} b \log n_j \\ &\leq b|C'| \log n \end{aligned}$$

其中 $b \geq 1$ 为常数。但是，由于 C' 是一个集合覆盖，因此

$$\sum_{i \in U} c(i) \leq \sum_{S_j \in C'} c(S_j)$$

于是，

$$|C| \leq b|C'| \log n$$

由这个事实可得以下结论。

定理13.15 SET-COVER问题的优化版本有 $O(\log n)$ 的近似多项式时间的算法，用于找出集合的集的覆盖，这些集合的并集为大小为 n 的全覆盖。

626

13.5 回溯法和分枝限界法

在上面几节中，我们证明了许多问题是NP完全问题。因此，除非 $P = NP$ ，否则大多数科学家认为这个式子不成立。不可能在多项式时间内求解其中的任意问题。然而，现实生活中的应用引出了其中的许多问题，它们需要找到解决的方法，即使要花费较长时间对它们进行求解。因此，在这一节里，我们论述处理NP完全性的技术，这些技术在实际中很有前景。这些技术使我们设计出在合理时间内找到求解困难问题的算法。在这一节里，研究回溯法（backtracking）和分枝限界法（branch-and-bound）。

13.5.1 回溯法

回溯设计模式总是建立某个困难问题 L 的算法。这个算法系统地查找一个较大的、甚至可能是指数级大小的集合。查找策略通常被优化成避免查找 L 的问题实例中的对称部分，并且遍历查找空间以便找到 L 的一个“容易”解（如果这样的解存在）。

回溯（backtracking）技术利用了许多NP完全问题具有的内在结构。给定一个多项式大小的证书，NP中的一个问题接受实例 x ，可在多项式时间内得到验证。常常是这个证书由“选择”集合组成。如赋给布尔变量集的一些值、包含在特殊集合中的图的顶点子集，或者包含在背包中的物品集。同样，证书验证常常涉及一个简单的测试，用于测试证书是否证实了 x 的一个成功配置，如满足公式、覆盖图中所有边、或符合某一性能准则。在这样的情况下，可以利用算法13-3中给出的回溯算法，系统地查找问题的一个解（如果存在这样的问题）。

算法13-3 回溯算法的模板

```

算法 Backtrack( $x$ ):
  输入: 困难问题的问题实例 $x$ 
  输出:  $x$ 的解, 或者如果不存在解, 则输出“无解”
   $F \leftarrow \{(x, \emptyset)\}$        $\{F$ 是子问题配置的“边界”集 $\}$ 
  while  $F \neq \emptyset$  do
    从 $F$ 中选择最有“希望”的配置 $(x, y)$ 
    通过做出另外一组较少的选择, 扩展 $(x, y)$ 
    设 $(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k)$ 是新配置集
    for 每个新配置 $(x_i, y_i)$  do
      对 $(x_i, y_i)$ 进行简单的一致性检查
      if 检查返回“找到解” then
        return 从 $(x_i, y_i)$ 导出的解
      if 检查返回“终结点” then
        丢弃配置 $(x_i, y_i)$        $\{\text{回溯}\}$ 
      else
         $F \leftarrow F \cup \{(x_i, y_i)\}$        $\{(x_i, y_i)$ 开始了一条有希望的查找路径 $\}$ 
    return “无解”
  
```

回溯算法遍历可能的“查找路径”，定位问题解或“终结点”。这条路径末尾的配置由数对 (x, y) 组成，其中 x 是要被求解的其余子问题， y 是从原问题实例得到这个子问题做出的选择集合。起初，给定回溯算法的数对 (x, \emptyset) ，其中 x 是原问题实例。任何时候只要回溯算法找到一个配置 (x, y) ，如果不论做出其他什么选择都不能导致有效解，那么从这个配置删除掉所有未来的查找，并“回溯”到另一个配置。事实上，回溯算法即由此得名。

1. 填充细节

为了把回溯策略转变成实际的算法，仅需要填充以下细节：

- 从边界集 F 定义选择最有“希望”的候选配置的一种方法。
- 确定把配置扩展成为子问题配置的一种方法。这个扩展过程从初始配置 (x, \emptyset) 开始，原则上应该能产生所有可行配置。
- 描述如何对配置 (x, y) 进行简单的一致性检查，它返回“找到解”、“终结点”或“继续”。

如果 F 是一个栈，那么得到深度优先查找配置空间。事实上，在这种情况下，甚至可以利用递归自动实现 F 为栈。另一方面，如果 F 是一个队列，那么得到广度优先查找配置空间。可以想象实现 F 的其他数据结构，但只要能够表示出每次迭代从 F 中选出最有“希望”的配置，就可得到回溯算法。

为使这个方法更具体，通过把回溯技术应用于CNF-SAT问题来观察它是如何工作的。

628

2. CNF-SAT的回溯算法

在CNF-SAT问题中，给定布尔公式 S 的合取范式（CNF），并询问 S 是否是可满足的。为了设计CNF-SAT问题的回溯算法，对 S 中的变量进行系统、尝试性的指派，并观察这样的指派是使 S 最终为1，还是为0，或者产生一个新的公式 S' ，对此公式可以继续尝试性的指派。因此，算法中的一个配置由对 (S', y) 组成，其中 S' 是CNF中的布尔公式， y 是不在 S' 中的布尔变量的指派值，使 S 中的这些指派导致公式 S' 。

为了形式化回溯算法，那么需要给出回溯算法这三部分中每部分的细节。给定配置边界 F ，做出最有“希望”的选择，它是具有最小子句的子公式 S' 。这个公式是 F 中所有公式中最受约束的公式；因此，期望它能最快速地命中终结点（如果这的确是它的目的地）。

然后考虑如何从子公式 S' 产生子问题。找出 S' 中的最小子句 C ，选择出现在 C 中的一个变量 x_i 。然后分别建立关于指派 $x_i = 1$ 和 $x_i = 0$ 的两个新的子问题。

最后，必须针对 S' 中变量 x_i 的指派值，考虑如何进行一致性检查。基于 x_i 指派的值0或1（取决于所做的选择），归约从包含 x_i 的子句开始。如果这个归约导致只有一个文字的新子句 x_j 或 \bar{x}_j ，还要对 x_j 进行相应值的指派，使得这个单文字的子句是满足的。然后处理结果公式，传播 x_j 的指派值。如果这个新公式依次导致一个单文字的新公式，重复这个过程，直到没有单文字的子句。如果在任何时刻，发现存在矛盾（即子句 x_i 与 \bar{x}_i ，或空子句），那么返回“终结点”。如果把子公式 S' 一直归约到常数1，那么返回“找到解”，以及到此点对所有变量所做的指派。否则，导出一个新公式 S'' ，满足每个子句至少有两个文字，以及从原始公式 S 到 S'' 所做的值指派。称这个操作为向 S' 中的 x_i 传播一个指派值的归约（reduce）操作。

把这些片断结合成为一个模板，得到解CNF-SAT问题的回溯算法。这个算法像我们预期的那样快。一般而言，这个算法最坏情况下的运行时间仍然是指数时间，但回溯可以起到加速作用。如果给定公式中的每个子句至多有两个文字，那么这个算法的运行时间为多项式时间（见习题C-13.4）。

629

3. Java示例：SUBSET-SUM的回溯算法解

为了更具体一些，考虑NP困难的SUBSET-SUM问题的Java回溯算法解，回忆13.3.4节的SUBSET-SUM问题，给定 n 个整数的集合 S 以及一个整数 k ，并询问集合 S 中是否存在求和为 k 子集。

为使确定一个配置是否是终结点的判定更容易，假设 S 中的整数按照非降次序排列为 $S = \{a_0, a_1, \dots, a_{n-1}\}$ 。给定这个选择，可以如下定义回溯算法的三个主要部分：

(1) 对于选择最有“希望”的候选配置的方法，与许多回溯算法中所做的相同，即利用递归对配置空间进行深度优先查找。因此，方法栈自动记录未探索的配置。

(2) 对于把配置 (x, y) 扩展成子问题配置所确定方式的过程, 按照 S 中整数的次序简单地向下进行。因此, 若有一个已经考虑的子集 $S_i = \{a_0, a_1, \dots, a_i\}$ 的配置, 通过是否利用 a_{i+1} , 只需产生两种可能的配置。

(3) 回溯算法的最后部分是执行一致性检查, 对返回“找到解”、“终结点”或者“继续”进行测试。为使测试简便, 再次利用 S 中整数有序的事实。假定已经考虑子集 $S_i = \{a_0, a_1, \dots, a_i\}$ 中的整数, 现在考虑 a_{i+1} 。在这一点上, 执行两方面的一致性测试。设 k_i 表示 S_i 中元素和, 这些元素经过尝试已被选择。设其余整数和为

$$r_{i+1} = \sum_{j=i+1}^{n-1} a_j$$

测试的第一部分是确认

$$k \geq k_i + a_{i+1}$$

如果这个条件不成立, 那么上式右边“超过” k 。但由于 S 有序, 对于 $j > i+1$ 的任何 a_j , 上式右边也会超过 k ; 因此, 这种情况为终结点。测试的第二部分是确认

$$k \leq k_i + r_{i+1}$$

如果这个条件不成立, 那么上式右边“不足” k , 且用 S 中的其余整数也不会使上式右边达到 k ; 因此, 这种情况也为终结点。如果两个测试都成立, 那么考虑 a_{i+1} 并且继续这一过程。

630

代码段13-1中给出了这个方法的Java代码。

代码段13-1 SUBSET-SUM问题的回溯算法的Java实现

```
/**
 * Method to find a subset of an array of integers summing to k, assuming:
 * - the array a is sorted in nondecreasing order,
 * - we have already considered the elements up to index i,
 * - the ones we have chosen add up to sum,
 * - the set that are left sum to remain.
 * The function returns "true" and prints the subset if it is found.
 * Should be first called as findSubset(a,k,-1,0,t), where t is total.
 */
public static boolean findSubset(int[] a, int k, int i, int sum, int remain) {
    /* Test conditions for expanding this configuration */
    if (i+1 >= a.length) return false; // safety check that integers remain
    if (sum + remain < k) return false; // we're undershooting k
    int next = a[i+1]; // the next candidate integer
    if (sum + next > k) return false; // we're overshooting k
    if (sum + next == k) { // we've found a solution!
        System.out.print(k + "=" + next); // begin printing solution
        return true;
    }
    if (findSubset(a, k, i+1, sum+next, remain-next)) {
        System.out.print(" + " + next); // solution includes a[i+1]
        return true;
    }
    else // backtracking - solution doesn't include a[i+1]
        return findSubset(a, k, i+1, sum, remain);
}
```

注意我们递归定义查找方法findSubset, 并且利用递归进行回溯, 当找到问题的解时, 递归

输出问题的解。因为将数组作为引用传递到基址中，每次递归调用的时间为 $O(1)$ 。因此，这个方法最坏情况下的运行时间为 $O(n^2)$ ，其中 n 是考虑的整数个数。当然，我们希望对于回溯法的典型问题实例，可以避免最坏情况，或者快速找到一个解，或利用终结点的条件清除那些无用配置。

如果能够切实地确信总是正确调用findSubset方法，则可以对其做出很小的改进，即可以减少对所考虑的剩余整数的安全测试。如果没有要考虑的整数，那么 $\text{remain} = 0$ 。因此， $\text{sum} + \text{remain} < k$ ，因为如果超过 k 或者恰好为 k ，则可以更早终止查找这条路径上的解。然而，尽管在理论上可以减少对整数所做的测试，但考虑到安全因素在代码中保留了这个测试，以防某人不正确调用我们的方法。

631

13.5.2 分枝限界法

回溯算法对于判定问题是有效的，但它并非设计用于优化问题。对于关联实例 x 的证书 y ，还要满足一些可行条件。我们还有一个希望最小化或最大化的代价函数 $f(x)$ （不失一般性，假设代价函数应该最小化）。然而，可以对回溯算法进行扩展，使其适合优化问题，由此导出的算法设计方法称为分枝限界法（branch-and-bound）。

算法13-4中给出的分枝限界设计模式包括所有回溯法中的要素，只不过它不是在找到解之后停止查找整个空间，而是继续进行，直到找到最好的解。此外，算法有一个计分机制，在每次迭代中总是选择最有希望的配置去探索。由于这个方法，分枝限界法有时称为最佳首次查找（best-first search）策略。

算法13-4 分枝限界算法的模板。这个算法假设存在函数 $lb(x_i, y_i)$ ，它返回从配置 (x_i, y_i) 导出任何解的代价的下界

```

算法 Branch-and-Bound( $x$ ):
  输入: 难解的优化（最小）问题的问题实例 $x$ 
  输出:  $x$ 的优化解，或如果不存在解则输出“无解”
   $F \leftarrow \{(x, \emptyset)\}$            { $F$ 是子问题配置的“边界”集}
   $b \leftarrow \{(+\infty, \emptyset)\}$    {当前最佳解的代价和配置}
  while  $F \neq \emptyset$  do
    从 $F$ 中选择最有“希望”的配置 $(x, y)$ 
    扩展 $(x, y)$ ，产生新配置 $(x_1, y_1), \dots, (x_k, y_k)$ 
    for 每个新配置 $(x_i, y_i)$  do
      对于 $(x_i, y_i)$ 进行简单的一致性检查
      if 检查返回“找到解” then
        if  $(x_i, y_i)$ 的解的代价 $c$ 比 $b$ 好 then
           $b \leftarrow (c, (x_i, y_i))$ 
        else
          丢弃配置 $(x_i, y_i)$ 
      if 检查返回“终结点” then
        丢弃配置 $(x_i, y_i)$            {回溯}
      else
        if  $lb(x_i, y_i)$ 小于 $b$ 的代价 then
           $F \leftarrow F \cup \{(x_i, y_i)\}$    { $(x_i, y_i)$ 开始一条有希望的查找路径}
        else
          丢弃配置 $(x_i, y_i)$            {“限界”剪枝}
  return  $b$ 
  
```

632

为了给出总是选择“最有希望”配置的优化准则，扩展回溯算法的三个假设，再加上另外一个条件：

- 对于任何配置 (x, y) ，假设有一个函数 $lb(x, y)$ ，它返回由此配置导出任一解代价的一个下界。

对 $lb(x, y)$ 的唯一严格要求是它必须小于或等于任一导出解的代价。但是根据分枝限界的描述应该显而易见，如果这个下界更准确，那么算法的效率就会提高。

1. TSP的分枝限界算法

为使分枝限界方法更具体，考虑如何把它应用于求解旅行推销员（TSP）问题的优化版本。在这个问题的优化版本中，给定图 G ，及为 G 中的每条边 e 定义的代价函数为 $c(e)$ ，希望找到访问 G 中的每个顶点并返回到其起始顶点的具有最小总代价的路径。

可以设计TSP的一个算法，对于每条边 $e = (v, w)$ ，计算开始于 v 、结束于 w 的具有最小代价的路径，同时按照一种方法，遍历 G 中所有其他的顶点。应用分枝限界技术找到这样一条路径。在分枝限界算法的每次循环中，在当前路径上一次增加一个顶点，生成 $G - \{e\}$ 中从 v 到 w 的路径。

- 构造从 v 开始的部分路径 P 后，只考虑用不在 P 中的顶点增大 P 。
- 如果不在 P 中的顶点在 $G - \{e\}$ 中不连通，则把这条部分路径 P 看作“终结点”。
- 利用 P 中边上的总代价以及 $c(e)$ 定义下界函数 lb 。这必定是从 e 和 P 构建的任何路径的一个下界。

此外，对于 G 中一条边 e 的算法运行完成之后，可以利用到目前为止测试过的所有边上所找到的最好路径，而不是在 $+\infty$ 处重新开始当前最好解 b 。最坏情况下结果算法的运行时间仍然为指数时间，但在实际中避免了大量不必要的计算。TSP问题非常有趣，它也有许多应用，因而如果输入图中的顶点数不太多，这样一个解在实际中是有用的。此外，还有许多其他启发式搜索方法可用于查找一条优化的TSP路径，但这些超出了本书的范围。

633

2. KNAPSACK问题的分枝限界解

为了更具体一些，我们描述KNAPSACK问题分枝限界解的Java实现。在这个问题的优化版本中，给定编号为 $0 \sim n-1$ 的物品集合 S 。每个物品 i 的大小为整数 s_i ，价值为 w_i 。给定整型参数 $size$ ，并询问是否存在 S 的子集 T ，满足 $\sum_{i \in T} s_i \leq size$ ，且使 T 中物品的总价值 $worth = \sum_{i \in T} w_i$ 达到最大。

我们借助贪心法的某些思想求解这个KNAPSACK问题（5.1.1节），即假设 S 中的物品按照 w_i/s_i 的非降序排列。我们按照这个次序处理物品，以便按递减收益考虑物品，从具有最大收益的元素开始，即从每单位大小价值最大的物品开始。用 S 中按这个次序排列的前 i 个物品的子集 S_i 定义配置。 S_i 中物品的下标在 $0 \sim i-1$ 之间（ S_0 为空配置，下标为 -1 ）。

首先把 S_0 的配置放进优先队列 P 中。然后在分枝限界算法的每次迭代中，选择 P 中最有希望的配置 c 。如果 i 是 c 的最后所考虑物品的下标，那么扩展 c ，变成两个配置，一个配置中包括 $i+1$ ，另一个配置中不包括它。注意满足大小约束条件的每个配置都是KNAPSACK问题的有效解。因此，如果两个配置中的一个有效，且优于当前找到的最好解，则将当前最好解更新为这两个中的较好者，并继续进行循环。

当然，为了选择最有希望的配置，需要一种根据配置可能值对它们进行计分的方式。在KNAPSACK问题中，我们对最大化价值感兴趣，而不是对最小化代价感兴趣，通过计算配置可能价值的一个上界作为配置的计分。确切地讲，给定配置 c ，已经考虑过的物品的下标为 $0 \sim i$ ，从 c 的总价值 w_c 开始，取 S 中剩余物品得到分数KNAPSACK问题的解，用来扩充 c ，得到向 c 中添加的价值，作为计算 c 的一个上界。回忆 S 中的物品已经按照 w_i/s_i 非降序排列，设 k 是满足 $\sum_{j=i+1}^k s_j \leq size - s_c$ 的最大下标，其中 s_c 是已在配置 c 中的所有物品的大小。下标为 $i+1 \sim k$ 的物品是完全适应在背包中的最好剩余物品。为了计算 c 的上界，那么考虑把这些元素都添加到 c 中，再加上物品 $k+1$ 适应背

包的尽可能多的那部分（如果存在），即 c 的上界定义如下：

$$\text{upper}(c) = w_c + \sum_{j=i+1}^k w_j + \left(\text{size} - s_c - \sum_{j=i+1}^k s_j \right) \frac{w_{k+1}}{s_{k+1}}$$

如果 $k = n-1$ ，那么假设 $w_{k+1}/s_{k+1} = 0$ 。

基于这个方法，代码段13-2、代码段13-3和代码段13-4中给出了KNAPSACK问题的Java分枝限界解。634

代码段13-2 KNAPSACK问题的分枝限界解的引擎方法。没有显示出类DoublePriorityQueue，它是一个优先队列，专门用于保存具有Double关键字的对象。注意优先队列中所用的关键字值为负，因为我们对最大化价值感兴趣，而不是对最小化代价感兴趣。同样注意到当扩展一个配置，而没有添加下一个元素时，并不检查这是否是一个更好的解，因为它的价值与其父结点相同

```
/**
 * Method to find an optimal solution to KNAPSACK problem, given:
 * - s, indexed array of the sizes
 * - w, index array of element worth (profit)
 * - indexes of s and w are sorted by w[i]/s[i] values
 * - size, the total size constraint
 * It returns an external-node Configuration object for optimal solution.
 */
public static Configuration solve(int[] s, int[] w, long size) {
    /* Create priority queue for selecting current best configurations */
    PriorityQueue p = DoublePriorityQueue();
    /* Create root configuration */
    Configuration root = new Configuration(s,w,size);
    double upper = root.getUpperBound(); // upper bound for root
    Configuration curBest = root; // the current best solution
    p.insertItem(new Double(-upper), root); // add root configuration to p
    /* generate new configurations until all viable solutions are found */
    while (!p.isEmpty()) {
        double curBound = -((Double)p.minKey()).doubleValue(); // we want max
        Configuration curConfig = (Configuration) p.removeMin();
        if (curConfig.getIndex() >= s.length-1) continue; // nothing to expand
        /* Expand this configuration to include the next item */
        Configuration child = curConfig.expandWithNext();
        /* Test if new child has best valid worth seen so far */
        if ((child.getWorth() > curBest.getWorth()) && (child.getSize() <= size))
            curBest = child;
        /* Test if new child is worth expanding further */
        double newBound = child.getUpperBound();
        if (newBound > curBest.getWorth())
            p.insertItem(new Double(-newBound), child);
        /* Expand the current configuration to exclude the next item */
        child = curConfig.expandWithoutNext();
        /* Test if new child is worth expanding further */
        newBound = child.getUpperBound();
        if (newBound > curBest.getWorth())
            p.insertItem(new Double(-newBound), child);
    }
    return curBest;
}
```

代码段13-3 构造和扩展配置的Configuration类及其方法。这些用在KNAPSACK问题的分枝限界解中。代码段12-4是这个类的其余内容

```
class Configuration {
    protected int index; // index of the last element considered
    protected boolean in; // true iff the last element is in the tentative sol'n
    protected long worth; // total worth of all elements in this solution
    protected long size; // total size of all elements in this solution
    protected Configuration parent; // configuration deriving this one
    protected static int[] s;
    protected static int[] w;
    protected static long bagSize;
    /** The initial configuration - is only called for the root config. */
    Configuration(int[] sizes, int[] worths, long sizeConstraint) {
        /* Set static references to the constraints for all configurations */
        s = sizes;
        w = worths;
        bagSize = sizeConstraint;
        /* Set root configuration values */
        index = -1;
        in = false;
        worth = 0L;
        size = 0L;
        parent = null;
    }
    /** Default constructor */
    Configuration() { /* Assume default initial values */ }
    /** Expand this configuration to one that includes next item */
    public Configuration expandWithNext() {
        Configuration c = new Configuration();
        c.index = index + 1;
        c.in = true;
        c.worth = worth + w[c.index];
        c.size = size + s[c.index];
        c.parent = this;
        return c;
    }
    /** Expand this configuration to one that doesn't include next item */
    public Configuration expandWithoutNext() {
        Configuration c = new Configuration();
        c.index = index + 1;
        c.in = false;
        c.worth = worth;
        c.size = size;
        c.parent = this;
        return c;
    }
}
```

636

代码段13-4 代码段13-3中Configuration类的支持方法。它用在KNAPSACK问题的分枝限界解中。计算上界的方法尤其重要

```
/** Get this configuration's index */
public long getIndex() {
    return index;
}
/** Get this configuration's size */
public long getSize() {
```

```

    return size;
}
/** Get this configuration's worth */
public long getWorth() {
    return worth;
}
/** Get this configuration's upper bound on future potential worth */
public double getUpperBound() {
    int g; // index for greedy solution
    double bound = worth; // start from current worth
    long curSize=0L;
    long sizeConstraint = bagSize - size;
    /* Greedily add items until remaining size is overflowed */
    for (g=index+1; (curSize <= sizeConstraint) && (g < s.length); g++) {
        curSize += s[g];
        bound += (double) w[g];
    }
    if (g < s.length) {
        bound -= w[g]; // roll back to worth that fit
        /* Add fractional component of the extra greedy item */
        bound += (double) (bagSize - size)*w[g]/s[g];
    }
    return bound;
}
/** Print a solution from this configuration */
public void printSolution() {
    Configuration c = this; // start with external-node Configuration
    System.out.println("(Size,Worth) = " + c.size + ", " + c.worth);
    System.out.print("index-size-worth list = [");
    for (; c.parent != null; c = c.parent) // march up to root
        if (c.in) { // print index, size, and worth of next included item
            System.out.print("(" + c.index);
            System.out.print(", " + s[c.index]);
            System.out.print(", " + w[c.index] + ")");
        }
    System.out.println("]");
}
}

```

637

13.6 习题

基础题

- R-13.1 Amongus教授证明了判定问题 L 是可归约到一个NP完全问题 M 的多项式时间。此外，在经过80页密密麻麻的数学推理后，只证明了 L 可在多项式内求解。它证明了 $P = NP$ 吗？说明原因。
- R-13.2 利用一个真值表把布尔公式 $B = (a \leftrightarrow (b + c))$ 转换成等价的CNF公式。显示真值表和中间DNF公式 \bar{B} 。
- R-13.3 证明问题SAT，取任意布尔公式 S 作为输入，问 S 是否是可满足的，它是否是NP完全问题。
- R-13.4 考虑问题DNF-SAT，取析取范式(DNF)的布尔公式 S 作为输入，问 S 是否是可满足的。描述DNF-SAT的一个确定性多项式时间的算法。
- R-13.5 考虑问题DNF-DISSAT，取析取范式(DNF)的布尔公式 S 作为输入，问 S 是否是不可满足的，即存在 S 中布尔变量的指派值，使其计算结果为0。证明DNF-DISSAT是NP完全问题。
- R-13.6 把布尔公式 $B = (x_1 \leftrightarrow x_2) \cdot (\bar{x}_3 + x_4 x_5) \cdot (\bar{x}_1 x_2 + x_3 \bar{x}_4)$ 转换成CNF公式。
- R-13.7 证明CLIQUE问题是NP完全问题。
- R-13.8 给定CNF公式 $B = (x_1) \cdot (\bar{x}_2 + x_3 + x_5 + \bar{x}_6) \cdot (x_1 + x_4) \cdot (x_3 + \bar{x}_5)$ ，说明把 B 归约为等价3SAT问题的输入。

- R-13.9 给定 $B = (x_1 + \bar{x}_2 + x_3) \cdot (x_4 + x_5 + \bar{x}_6) \cdot (x_1 + \bar{x}_4 + \bar{x}_5) \cdot (x_3 + x_4 + x_6)$, 通过归约布尔公式 B 的 3SAT 形式, 构造 VERTEX-COVER 的一个实例。
- R-13.10 画出一个具有 10 个顶点、15 条边且顶点覆盖大小为 2 的图。
- R-13.11 画出一个具有 10 个顶点、15 条边且有一个大小为 6 的团的图。
- R-13.12 Amongus 教授设计了一个算法, 取 n 个顶点的图作为输入, 用 $O(n^k)$ 时间确定 G 是否包含大小为 k 的团。Amongus 教授会由于证明 $P = NP$ 而应该得到图灵奖吗? 说明原因。
- R-13.13 存在和数为 100 的数 $\{23, 59, 17, 47, 14, 40, 22, 8\}$ 的子集吗? 如果和数为 130 呢? 说明原因。
- R-13.14 证明 SET-COVER 问题是 NP 完全问题。
- R-13.15 证明 SUBSET-SUM 问题是 NP 完全问题。
- R-13.16 画出一个具有 10 个顶点、20 条边且存在哈密顿回路的图。另外, 画出一个具有 10 个顶点、20 条边不具有哈密顿回路的图。
- R-13.17 平面上两个点 (a, b) 和 (c, d) 之间的曼哈顿距离 (Manhattan distance) 为 $|a-c| + |b-d|$ 。利用曼哈顿距离定义每对顶点之间的代价, 找出以下点集的最优旅行推销员路径:
 $\{(1, 1), (2, 8), (1, 5), (3, -4), (5, 6), (-2, -6)\}$
- R-13.18 详细描述对于具有 n 个顶点、 m 条边的图, 如何用 $O(n+m)$ 的时间实现算法 13-1。你可以利用传统的操作计数作为运行时间的测度。
- R-13.19 描述算法 13-2 有效实现的细节, 并分析它的运行时间。
- R-13.20 给出一个至少有 10 个顶点的图 G 的例子, 满足上面给出的 VERTEX-COVER 的贪心 2-近似算法保证产生次优顶点覆盖。
- R-13.21 给出一个完全加权图 G , 使其边上的权值满足三角不等式, 但 TSP 问题的基于 MST 的近似算法不能找到最优解。
- R-13.22 给出 CNF-SAT 问题回溯算法的伪代码描述。
- R-13.23 给出递归回溯算法的伪代码描述。假设查找策略应该按照深度优先方式访问配置。
- R-13.24 给出 TSP 问题的分枝限界算法的伪代码描述。
- R-13.25 13.5.2 节中描述的求解 KNAPSACK 问题的分枝限界程序利用布尔标志确定一件物品是否包括在解中。证明这个标志是多余的。也就是说, 即使去掉这个域, 也存在一种方式 (不用增加额外的域) 可用于表明物品是否包括在解中。

创新题

- C-13.1 证明可用多项式时间确定性地模拟一个运行时间为多项式时间的非确定性算法 A , 并且至多调用 $O(\log n)$ 次 choose 方法, 其中 n 是算法 A 的输入大小。
- C-13.2 证明 P 中的每种语言 L 是可归约到语言 $M = \{5\}$ 的多项式时间, 即语言简单地询问输入的二进制编码是否为 5?
- C-13.3 证明如何构造布尔电路 C , 使得如果只构造 C 的输入的变量, 然后设法建立与 C 等价的布尔公式, 那么构造的公式比 C 的编码大指数级。
 提示: 利用递归对每次所用的子表达式反复加倍。
- C-13.4 证明对于 13.5.1 节给出的 CNF-SAT 问题的回溯算法, 如果给定布尔公式中的每个子句至多两个文字, 则它的运行时间为多项式时间, 即 2SAT 问题有多项式时间的解。
- C-13.5 考虑 CNF-SAT 问题的 2SAT 版本, 给定公式 S 中的每个子句只有两个文字。注意形如 $(a + b)$ 的子句可看作两个蕴涵式 $(\bar{a} \rightarrow b)$ 和 $(\bar{b} \rightarrow a)$ 。考虑由 S 所得的图 G , 使 G 中每个顶点关联 S 中的一个变量 x 或 \bar{x} 。对于每个与 $(\bar{a} \rightarrow b)$ 等价的子句, 在 G 中设一条从 \bar{a} 到 b 的有向边。证明 S 是不可满足的, 当且仅当存在变量 x , 满足 G 中存在从 x 到 \bar{x} 的路径及从 \bar{x} 到 x 的路径。由此规则, 导出解 CNF-SAT 的这个特例的一个多项式时间的算法。并分析你所设计算法的运行时间。
- C-13.6 假定一位智者给你一台魔力计算机 C , 给定 CNF 中的任何布尔公式 B , 它会利用一个步骤告诉你 B 是否是可满足的。证明如何利用 C 构造任一可满足公式 B 的可满足布尔变量的实际指派值。要做到这一点, 最坏情况下需要进行多少次调用?

- C-13.7 定义SUBGRAPH-ISOMORPHISM问题, 它以图 G 和另一个图 H 作为输入, 确定 H 是否是 G 的一个子图, 即存在从 H 中每个顶点 v 到 G 中每个顶点 $f(v)$ 的映射, 使得如果 (v, w) 是 H 中的一条边, 那么 $f(v, w)$ 是 G 中的一条边。证明SUBGRAPH-ISOMORPHISM是NP完全问题。
- C-13.8 定义INDEPENDENT-SET问题, 它以图 G 和一个整数 k 作为输入, 问 G 中是否包含顶点规模为 k 的独立集合, 即 G 中包含规模为 k 的顶点集合 I , 使对于 I 中任意顶点 v 和 w , G 中不存在边 (v, w) 。证明INDEPENDENT-SET是NP完全问题。
- C-13.9 定义HYPER-COMMUNITY问题, 它以 n 个网页集合和一个整数 k 作为输入, 确定是否存在 k 个网页, 它们全都包含指向彼此的超链接。证明HYPER-COMMUNITY是NP完全问题。
- C-13.10 定义PARTITION问题, 它以数的集合 $S = \{s_1, s_2, \dots, s_n\}$ 作为输入, 确定是否存在 S 的子集 T , 满足

$$\sum_{s_i \in T} s_i = \sum_{s_i \in S-T} s_i$$

即询问是否能够把数划分成两组, 使得这两组中的数之和相等。证明PARTITION是NP完全问题。

- C-13.11 证明有向图的HAMILTONIAN-CYCLE问题是NP完全问题。
- C-13.12 证明如果给定输入为一元编码, 则SUBSET-SUM问题有多项式时间的解, 即证明SUBSET-SUM不是强NP困难的。并分析你所设计算法的运行时间。
- C-13.13 证明如果给定输入为一元编码, 则KNAPSACK问题有多项式时间的解, 即证明KNAPSACK问题不是强NP困难的。并分析你所设计算法的运行时间。
- C-13.14 考虑TSP问题的一般优化版本, 可能的图不必满足三角不等式。证明对于任意固定值 $\delta \geq 1$, 一般TSP不存在多项式时间的 δ -近似算法, 除非 $P = NP$ 。
提示: 把HAMILTONIAN-CYCLE归约到这个问题。定义 n 个顶点输入图 G 的完全图 H 的代价函数, 使得也在 G 中的 H 中的每条边的代价为1, 而不在 G 中的 H 中的每条边有大于1的代价 δn 。
- C-13.15 考虑TSP的特例, 其中顶点对应平面上的点, 每条边 (p, q) 上定义的代价为 p 和 q 之间的欧几里得距离。证明最优路径中没有任何交叉边。
- C-13.16 导出HAMILTONIAN-CYCLE问题的一个有效回溯算法。
- C-13.17 导出背包判定问题的一个有效回溯算法。
- C-13.18 导出背包优化问题的一个有效分枝限界算法。
- C-13.19 导出求解TSP优化问题的分枝限界算法的一个新的下界函数 lb 。你的函数应该总是大于或等于13.5.2节中使用的 lb 函数, 但仍是一个有效的下界函数。描述一个 lb 严格大于13.5.2节中使用的 lb 函数的例子。

640

程序设计

- P-13.1 设计和实现CNF-SAT问题的回溯算法。针对大量2SAT和3SAT的问题实例, 比较算法的运行时间。
- P-13.2 设计和实现TSP问题的分枝限界算法。利用至少两种不同下界函数 lb 的定义, 测试它们的有效性。
- P-13.3 分组设计和实现TSP问题的分枝限界算法和多项式时间的近似算法。对于用欧几里得距离定义每对点之间代价的平面上的点集, 测试这两种实现找出旅行推销员路径的效率和效果。
- P-13.4 实现HAMILTONIAN-CYCLE的回溯算法。对于不同的 n 值, 边数分别为 $2n$ 、 $\lceil n \log n \rceil$ 、 $10n$ 和 $20 \lceil n^{1.5} \rceil$, 测试算法找出哈密顿回路的有效性。
- P-13.5 对利用动态规划和回溯法解SUBSET-SUM问题, 进行经验性的比较。
- P-13.6 对利用动态规划和分枝限界法解KNAPSACK问题, 进行经验性的比较。

641

13.7 本章注记

Lewis和Papadimitriou[133]、Savage[177]和Sipser[187]的著作中讨论了计算模型。

本章给出的Cook-Levin定理的简洁证明由Cormen、Leiserson和Rivest[55]的证明改编而来。Cook的原始定理[53]证明了CNF-SAT是NP完全问题, Levin的原始定理[131]针对层叠问题。称定理13.2为“Cook-Levin”定理是为了纪念这两篇创新性的论文, 因为他们的证明与定理13.2给出

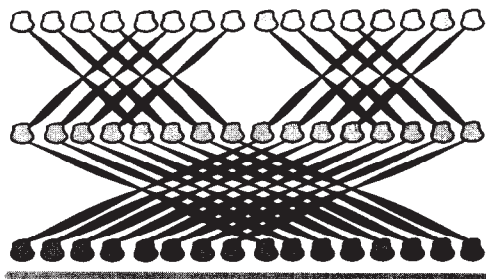
的证明思路相同。Karp[113]证明了更多的问题是NP完全问题，随后其他数百个问题被证明是NP完全问题。Garey和Johnson[76]对NP完全性做了很好的讨论，并对重要的NP完全问题和NP困难问题进行了分类。

本章给出的归约利用局部替换法和限制法，它们在计算机科学文献中非常常见；例如，参见Garey和Johnson[76]、Aho、Hopcroft和Ullman[7]。证明VERTEX-COVER是NP完全问题的分量设计证明方法是对Garey和Johnson证明[76]的改编。如同在证明HAMILTONIAN-CYCLE是NP完全问题的分量设计方法中一样，它自身是Karp[113]所提出的两个归约的组合。证明SUBSET-SUM是NP完全问题的分量设计证明方法是对Cormen、Leiserson和Rivest[55]证明的改编。

在Lewis和Papadimitriou[133]及Brassard和Bratley[38]讨论了回溯法和分枝限界法之后，将它们的讨论进行了模型化。回溯法可用于判定问题的求解，分枝限界法可用于优化问题的求解。然而，我们的讨论也受到Neapolitan和Naimipour[159]的影响，他们从另一个角度把回溯法看作是一种利用广度优先查找或最佳首次查找的启发式搜索方法，并用下界函数进行剪枝。回溯法技术自身可追溯到Golomb和Baumert[80]早期的工作。

对近似算法的一般讨论可在其他一些著作中找到，包括Hochbaum[97]及Papadimitriou和Steiglitz[165]的著作，以及Klein和Young[116]的部分章节。在Ibarra和Kim[106]给出的结论之后，Klein和Young[116]给出了KNAPSACK问题的PTAS模型。Papadimitriou和Steiglitz把VERTEX-COVER的2-近似算法归因于Gavril和Yannakakis。TSP特例的2-近似算法归因于Rosenkrantz、Stearns和Lewis[174]。SET-COVER的 $O(\log n)$ -近似算法及其证明得自Chvátal[46]、Johnson[109]和Lovász[136]的工作。

第14章



算法框架

本书大部分内容基于一个计算框架。这个框架的正式名是随机存储机 (RAM)，它由单一处理器及与它连接的可能无界的单一存储器组成。而且，本书重点关注接受一个输入，然后处理那个输入以产生一个输出的算法研究。这个框架对本书非常适用，能够建模算法设计者遇到的大多数计算。然而它有其局限性，对几个自然且用途较大的计算环境并不适用。本章将研究三个适用于这些环境的框架。

研究的第一个框架是对RAM模型的存储器组件的扩展。在这个称为外存 (external memory) 模型的框架中，我们试图更现实地建模出现在现代计算机中的存储器层次结构。尤其是试图建模存储器，把存储器划分成快速内存和慢速外存，前者访问时间快，但存储容量小；后者访问时间慢，但存储容量大。设计这种存储器层次结构的有效算法要求对假设所有存储器访问时间相同的技术进行一些修改。在这一章里，考察对查找算法和排序算法所做的某些特定改变，使这些算法有效适合于外存。

我们要考虑的传统RAM框架的另一种重要变体用于并行算法，即能够利用多个处理器共同解决一个问题的算法。在并行算法设计中，希望并行算法解能够通过尽可能接近处理器数的线性因子改进已知的串行算法解。我们研究几个重要的并行算法，包括算术运算、查找和排序的并行算法，寻找能够对RAM模型进行并行扩展使之适合于多个处理器的方式。

本章中考察的最后一个框架对“算法只是把输入映射到输出的函数”这种观点提出了挑战。在在线算法的框架中，把算法看作是必须处理一系列客户请求的服务器，这些请求随着时间的推移而得到处理。在检查和处理下一个请求之前，必须完全处理对请求的响应。由于在因特网上常常使用计算机处理来自远程用户的计算请求，从而产生了这个模型。设计这种模型的算法所面临的挑战是对一个请求做出的选择可能使处理将来的一个请求的时间要长得得多。为了分析一个在线算法选择的有效性，我们常常把它的行为与预先知道的客户请求序列的离线算法的行为进行比较。称这样的分析为竞争性分析，它与本书中用于分析把输入映射为输出的算法最坏情况下的时间复杂度形成一种自然类似。

644

14.1 外存算法

有一些计算机应用必须处理大量的数据，例如科学数据集的分析、金融交易的处理与数据的组织和维护（如电话目录）。事实上，必须被处理的数据量常常太大，以至于不能完全装入计算机的内存中。

1. 存储器层次结构

为了容纳大量数据集，计算机具有不同类型的存储器的层次结构（hierarchy），它们在其大小及与CPU的距离这些方面有所不同。与CPU距离最近的是CPU自身所用的内部寄存器。CPU对这些内部寄存器的访问速度非常快，这样的内部寄存器相对较少。层次结构中的第二层是高速缓冲（cache）存储器。这个存储器比CPU中的寄存器集要大得多，但CPU访问它所需的时间较长（甚至可能有多个高速缓冲存储器，CPU对这些存储器的访问速度逐渐减慢）。层次结构中的第三层是内存（internal memory），也称之为主存（main memory）、核心存储器（core memory）或随机存取器（RAM）。内存比高速缓存大得多，但访问所需时间较长。最后，层次结构的最高层是外存（external memory），通常由磁盘、CD或磁带组成。这个存储器非常大，但也非常慢。因此，可把计算机的存储器层次结构看作由4层组成，外存大小最大，内存次之，高速缓冲存储器第三，内部寄存器最小；但CPU对它们的访问速度正好相反，即对内部寄存器访问速度最快，对外存访问速度最慢（见图14-1）。

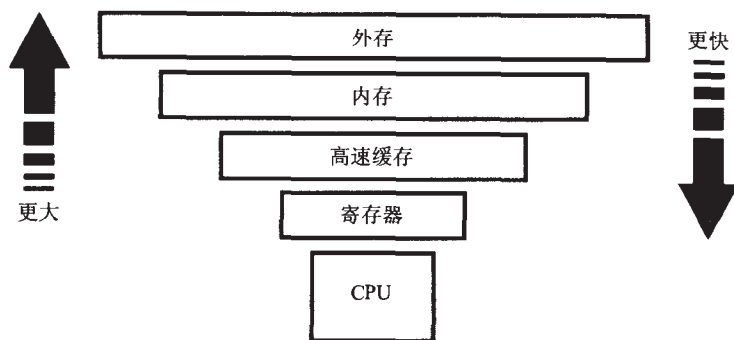


图14-1 存储器的层次结构

645

但是，在大多数应用中，只有两层确实重要——一层能够容纳问题中的所有数据；另一层恰好在这一层的下面。把数据项调进和调出能够容纳所有数据项的高层存储器在这种情况下通常会成为计算瓶颈。

2. 高速缓存和磁盘

特定的两层主要取决于我们试图求解问题的大小。对于能够完全装入主存中的问题，重要的两层是高速缓存和内存。访问内存的时间可能比访问高速缓存的时间长10~100倍。因此，希望大多数对存储器的访问能够在高速缓存中进行。另一方面，对于不能完全装入主存中的问题，重要的两层是内存和外存。在这里，这种差异甚至更显著，因为访问磁盘（通常是用于一般目的的外存设备）的时间一般比访问主存的时间长100 000~1 000 000倍。

考虑后一种情况，想象一位在巴尔的摩的学生向他在芝加哥的父母发送一条请求金钱的消息。如果这个学生向他的父母发送一条电子邮件消息，这条消息大约在5秒钟之内就能到达其父母的计算机上，这种通信模式可以看作CPU对内存访问的响应。对应于访问速度慢500 000倍的外存访问的一种通信模式是，该学生步行到芝加哥，将其请求告诉父母，如果他每天能够步行20英里，则需要花费大约一个月的时间。因此，我们应该尽可能少的访问外存。

本节讨论分层的存储器管理的一般策略，并介绍用于外存查找和排序的一些方法。

14.1.1 分层的存储器管理

大多数算法设计没有考虑存储器层次结构，尽管对于不同层的访问时间差别极大。的确，本

书中到目前为止描述的所有算法分析都假设访问所有存储器的时间相同。这个假设初看上去存在很大的疏忽——现在只在最后一章提出——但有两个基本理由说明为什么做出这种假设是合理的。

常常假设访问所有存储器的时间相同的第一个理由是，因为依赖设备的关于存储器大小的特定信息难以得到。事实上，不可能得到存储器大小的信息。例如，设计在不同计算机平台上运行的Java程序不能按照特定的计算机体系结构配置定义。如果确实有这个信息（本章后面将探讨如何利用这样的信息），我们的确可以利用特定体系结构的信息。但一旦在某种体系结构的配置上优化了软件，我们的软件就不再是独立于设备的。幸运的是，这样的优化并不总是必要的，主要是由于等时访问存储器假设的第二个证明。

646

等时访问存储器这一假设的第二个证明是操作系统设计者已经开发了快速访问大多数存储器的一般机制。这些机制基于多数软件具有的两个重要局部引用（locality-of-reference）性质：

- 时间局部性（temporal locality）：如果一个程序访问存储器中的某一位置，那么在不久的将来可能再次访问这个位置。例如，在几个不同的表达式中，利用计数器变量的值相当普遍，包括增加计数器的值。事实上，计算机体系结构师当中的普遍格言是“程序在其10%的代码中花费了90%的时间”。
- 空间局部性（spatial locality）：如果一个程序访问存储器中的某一位置，那么它可能访问这个位置附近的其他位置。例如，一个利用数组的程序可能按顺序或以接近顺序的方式访问这个数组中的位置。

计算机科学家和工程师已经执行了广泛的软件成型实验，证实了大多数软件具有局部引用的这两种性质。例如，用于扫描数组的for循环将展示这两种性质。

1. 缓存和分块

时间局部性和空间局部性反过来又引出了两层计算机存储系统的两种基本的设计选择（存在于高速缓存和内存之间的接口中，还存在于内存和外存之间的接口中）。

第一种设计选择称为虚拟存储（virtual memory）。这个概念由提供了和二级存储器以及把数据传递到主存中的空间大小相同的地址空间组成，数据编址位于二级存储器中。虚拟存储使程序员不受内存大小的限制。把数据调进主存中的概念称为缓存（caching），由时间局部性所推动。因为，通过把数据调进主存，希望在不久的将来，它能很快被再次访问，同时还能够快速响应对数据的所有请求。

第二种设计选择受到空间局部性的推动所致。确切地讲，如果访问存储在二级存储器位置 I 处的数据，那么就会把包含位置 I 的一大块连续位置调入主存（见图14-2）。这个概念称为分块（blocking），这是期望不久会访问位置 I 附近的其他二级存储器位置。在高速缓存和内存之间的接口中，常常称这样的块为高速缓存管线（cache line），在内存和外存之间的接口中，常常称这样的块为页面（page）。

647

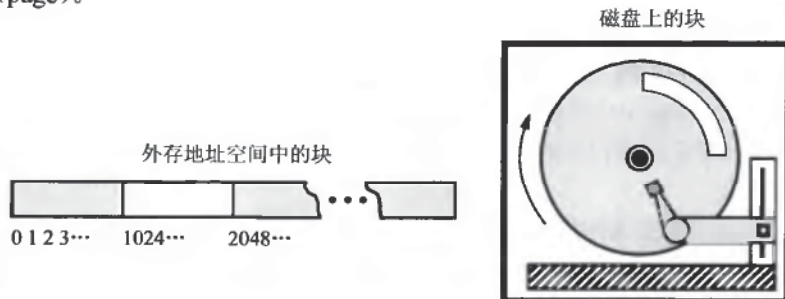


图14-2 外存中的块

顺便提一句，磁盘和CD-ROM驱动器的分块也受到这些硬件技术所具有的性质影响。对于磁盘或者CD-ROM的读取臂，需要相对长的时间定位读取位置。但是，一旦定位了读取臂，它可以快速读取许多连续位置，因为它所读取的介质旋转得非常快（见图14-2）。然而，即使没有这个动机，分块合理性也可由大多数程序具有的空间局部性得到完全证实。

因此，当用高速缓存和分块实现时，虚拟存储常常使我们感觉到二级存储器比其实际的速度还要快。然而仍然存在一个问题，主存要比二级存储器小得多。此外，由于内存系统利用分块，任何程序实例可能碰到这样一种情况，它向二级存储器请求数据，但是主存已经完全充满了块。为了完成这个请求，维护对高速缓存和分块的使用，在这种情况下必须从主存中删除某些块，用于来自二级存储器中的新块。决定如何进行页面替换的问题引出了许多有趣的数据结构和算法设计问题，在本节后面将会讨论。

2. 外部查找模型

我们提出的第一个问题是实现一个大型数据项集合的字典，这个数据项集合不能一次完全调入内存。请记住字典可以存储关键字-元素对（数据项），可对它进行插入、删除和基于关键字的查找操作。因为大型字典的主要应用之一是在数据库系统中，我们称二级存储器块为磁盘块（disk block）。同样，称二级存储器和主存中进行的块传输为磁盘传输（disk transfer）。即使利用这个术语，本节讨论的查找技术也可应用在主存是CPU高速缓存和二级存储器是主（内）存的情况下。利用这种基于磁盘的观点，是因为它是具体的且更具有普遍性。

648

14.1.2 (a, b)树和B树

由于在访问主存和访问磁盘的时间上存在巨大差异，维护外存中字典的主要目标是最小化查询或更新所需的磁盘传输次数。事实上，磁盘和内存之间速度上的差别是如此巨大，以至于如果能够避免多次磁盘传输，我们愿意执行对内存的大量访问。于是，通过对每次执行标准字典查找和更新操作的磁盘传输次数进行统计，来分析字典实现的性能。

首先考虑某些基于序列的外存效率低下的字典实现方法。如果用无序、双向链表实现表示字典的序列，那么插入和删除操作均需要 $O(1)$ 次传输，假设知道要被删除的数据项保存在哪个块中。但在最坏情况下查询需要 $\Theta(n)$ 次传输，因为进行的每次链跳可能访问不同的块。这个查找时间可以改进到 $O(n/B)$ 次传输（见习题C-14.1），其中 B 表示链表中适合块的结点数，但这样做性能仍然不佳。我们也可以用一个有序数组实现序列。在这种情况下，利用二分搜索进行一个查找需要 $O(\log_2 n)$ 次传输，这是一个良好的改进。但这种解决方案实现一个插入或删除在最坏情况下需要 $\Theta(n/B)$ 次传输，因为可能访问所有块以进行元素的上下移动。因此，序列字典实现不是外存有效的方法。

如果序列实现是效率低下的，那么也许我们应该考虑对数时间的内存策略，它利用平衡二叉树（例如，AVL树或红黑树）或具有对数平均查询时间和更新时间的其他查找结构（例如，跳跃表或伸展树）。这些方法把字典数据项存储在二叉树或图中的结点中。在最坏情况下，在其中一个结构中对每个结点的查询或更新会在不同的块中进行。因此，这些方法进行一次查询或更新操作最坏情况下都要求 $O(\log_2 n)$ 次传输。这是好的，但我们能做得更好。尤其是，只利用 $O(\log_B n) = O(\log n / \log B)$ 次传输就可以进行字典查询和更新。

1. 更好的方法

改进上述讨论的字典实现的外存性能的主要思想是，我们愿意执行多达 $O(B)$ 次内存访问，以避免一次磁盘传输，其中 B 表示块的大小。驱动磁盘的软硬件执行这个对内存的多次访问，只是为了把一个块引入内存，这仅仅是磁盘传输代价的一小部分。因此，为了避免耗时的磁盘传输， $O(B)$ 次对内存的高速访问所付的代价较小。

649

2. (a, b)树

为了减少查找时在访问内存和访问外存之间所产生的性能差异的重要影响,可以利用多路查找树(第3章)表示字典。这种方法导致对(2, 4)树数据结构的推广,称这个推广的结构为(a, b)树。

形式上,一棵(a, b)树是一棵多路查找树,每个结点的子结点数介于 $a \sim b$ 之间,且存储的数据项数介于 $a-1 \sim b-1$ 之间。在(a, b)树中进行查找、插入和删除的算法是(2, 4)树相应操作的直观推广。把(2, 4)树推广到(a, b)树的优点是,一般意义的树提供了灵活的查找结构,结点大小和各种字典操作的运行时间依赖于参数 a 和 b 。通过把 a 和 b 设置为关于磁盘块大小的合适参数,可以导出达到良好的外存性能的数据结构。

一棵(a, b)树((a, b) tree)是一棵多路查找树,其中 a 和 b 是整数,满足 $2 \leq a \leq (b+1)/2$,具有如下另外两个限制:

大小性质(size property): 每个内部结点至少有 a 个子结点(除非它是根),至多有 b 个子结点。

深度性质(depth property): 所有外部结点的深度相同。

定理14.1 存储 n 个数据项的一棵(a, b)树的高度为 $\Omega(\log n / \log b)$ 和 $O(\log n / \log a)$ 。

证明 设 T 是存储 n 个元素的一棵(a, b)树,并设 h 是 T 的高度。通过建立关于 h 的以下界限,证明定理是正确的:

$$\frac{1}{\log b} \log(n+1) \leq h \leq \frac{1}{\log a} \log \frac{n+1}{2} + 1$$

由大小和深度性质可知, T 中外部结点数 n'' 至少为 $2a^{h-1}$ 且至多为 b^h 。由定理3.3可知, $n'' = n + 1$ 。因此

$$2a^{h-1} \leq n+1 \leq b^h$$

对上面的不等式取以2为底的对数,可得

$$(h-1)\log a + 1 \leq \log(n+1) \leq h\log b$$

■

回忆在一棵多路查找树 T 中, T 中的每个结点 v 保存一个二级结构 $D(v)$,它自身是一个字典(3.3.1节)。如果 T 是一棵(a, b)树,那么 $D(v)$ 至多存储 b 个数据项。令 $f(b)$ 表示在 $D(v)$ 字典中进行一次查找的时间。在(a, b)树中进行查找的算法正像3.3.1节给出的多路查找树中的查找算法。因此,在 n 个数据项中的(a, b)树 T 中进行查找需要 $O\left(\frac{f(b)}{\log a} \log n\right)$ 的时间。注意如果 b 是一个常数(因而 a

650

也是常数),那么查找时间为 $O(\log n)$,这个结果独立于二级结构的特定实现。

(a, b)树的主要应用是作为存储在外存(例如,磁盘或CD-ROM)中的字典使用。也就是说,选择参数 a 和 b ,每个树结点占据一个磁盘块(如果想要简单统计块传输次数,设 $f(b) = 1$),使得访问磁盘次数达到最少。在这个环境中提供恰当的 a 和 b 的值引出了我们将简略讨论的称为B树的数据结构。然而,在描述这个结构之前,我们讨论如何在(a, b)树中进行插入和删除操作。

3. (a, b)树中的插入和删除

(a, b)树中的插入算法类似于(2, 4)树中的插入算法。当将一个数据项插入到一个 b -结点 v 中时,会出现上溢,因为这会使这个结点成为 $(b+1)$ -结点(回忆如果多路查找树中的一个结点有 d 个子结点,则称它是一个 d -结点)。为了修正上溢,通过把结点 v 的中间结点移到父结点中,并用 $\lceil (b+1)/2 \rceil$ -结点 v' 和 $\lfloor (b+1)/2 \rfloor$ -结点 v'' 代替结点 v ,完成对结点 v 的分裂。现在可以看出定义(a, b)树时,要求 $a \leq (b+1)/2$ 的原因。注意分裂的结果,需要构建二级结构 $D(v')$ 和 $D(v'')$ 。

(a, b) 树中的删除算法也类似于 $(2, 4)$ 树中的删除算法。当从一个不同于根的 a -结点 v 删除结点时, 会出现下溢, 这使 v 成为一个非法的 $(a-1)$ -结点。为了修正下溢, 可以对不是 a -结点的 v 的兄弟结点进行一次转移, 或者把 v 与是 a -结点的一个兄弟结点进行融合。由融合导致的新结点 w 是一个 $(2a-1)$ -结点。这里又看到要求 $a \leq (b+1)/2$ 的另一个原因。注意融合的结果, 需要构建二级结构 $D(w)$ 。表14-1显示了 (a, b) 树 T 实现的字典中主要操作的运行时间。

表14-1 (a, b) 树实现的字典中主要方法的时间复杂度。令 $f(b)$ 表示查找 b -结点的时间, $g(b)$ 表示分裂或融合一个 b -结点的时间。同时用 n 表示字典中的元素个数。空间复杂度为 $O(n)$

方 法	时 间
findElement	$O\left(\frac{f(b)}{\log a} \log n\right)$
insertItem	$O\left(\frac{g(b)}{\log a} \log n\right)$
removeElement	$O\left(\frac{g(b)}{\log a} \log n\right)$

651

表14-1中的界限基于以下假设和事实:

- 利用3.3.1节描述的数据结构表示 (a, b) 树 T , T 中结点的二级结构支持时间为 $f(b)$ 的查找, 分裂操作和融合操作的时间为 $g(b)$, 对于某些函数 $f(b)$ 和 $g(b)$, 在只统计磁盘传输次数的环境中可以达到 $O(1)$ 。
- 存储 n 个元素的 (a, b) 树 T 的高度至多为 $O((\log n)/(\log a))$ (定理14.1)。
- 在根结点和一个外部结点的路径上进行查找要访问 $O((\log n)/(\log a))$ 个结点, 每个结点上花费的时间为 $f(b)$ 。
- 传输操作所需时间为 $f(b)$ 。
- 分裂或融合操作所需时间为 $g(b)$, 为创建新结点 s 构建的二级结构大小为 $O(b)$ 。
- 在根结点和一个外部结点的路径上进行插入或删除要访问 $O((\log n)/(\log a))$ 个结点, 每个结点上花费的时间为 $g(b)$ 。

因此, 结论概括如下。

定理14.2 用 (a, b) 树实现 n 个数据项的字典, 其上支持的插入和删除操作所需时间为 $O((g(b)/\log a) \log n)$, 查询操作所需时间为 $O((f(b)/\log a) \log n)$ 。

4. B树

(a, b) 树数据结构的一个特殊版本是一种称为“B树”(见图14-3)的数据结构, 它是一个维护外存中字典的有效方法。阶(order)为 d 的B树是一棵 $a = \lceil d/2 \rceil$ 和 $b = d$ 的简单 (a, b) 树。因为上面讨论了 (a, b) 树的标准字典查询和更新方法, 我们这里的讨论只分析B树的外存性能。

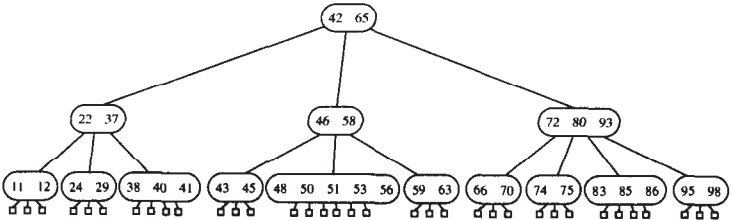


图14-3 阶为6的B树

652

5. 外存B树的参数化

对B树进行的最重要的观察是可以选择 d ，使得存储在一个结点中的 d 个子结点引用和 $(d-1)$ 个关键字适合于存放在磁盘的一个块中，即选择 d ，满足

$$d \text{ 是 } \Theta(B)$$

这种选择还蕴涵着在分析 (a, b) 树的查找和更新操作时，可以假设 a 和 b 是 $\Theta(B)$ 。同样，回忆我们主要对进行各种操作所需的磁盘传输次数感兴趣。因此， d 的选择还蕴涵着对于某个常数 $c \geq 1$ ，

$$f(b) = c$$

和

$$g(b) = c$$

因为每次访问一个结点进行查找或更新操作时，只需要进行一次磁盘传输，即 $f(b)$ 和 $g(b)$ 都为 $O(1)$ 。正如已经观察到的那样，每次查找或更新至多会考察树的每一层的 $O(1)$ 个结点。于是，在B树上进行任何字典查找或更新操作只需要

$$\begin{aligned} O(\log_{\lceil d/2 \rceil} n) &= O(\log n / \log B) \\ &= O(\log_B n) \end{aligned}$$

次磁盘传输。例如，插入操作在B树上向下进行，以定位插入新数据项的位置。如果由于增加子结点导致结点上溢（有 $d+1$ 个子结点），那么把这个结点分裂成分别有 $\lfloor (d+1)/2 \rfloor$ 和 $\lceil (d+1)/2 \rceil$ 个子结点的两个结点。然后在下一层向上重复这个过程，至多继续进行 $O(\log_B n)$ 层。同样，在一个删除操作中，从结点上删除一个数据项，结点可能下溢（有 $\lceil d/2 \rceil - 1$ 个子结点），那么要么从一个至少有 $\lceil d/2 \rceil + 1$ 个子结点的兄弟结点移动引用，要么需要把这个结点与它的兄弟结点进行融合（并在父结点处重复这个计算）。和插入操作一样，对于B树这个过程至多向上继续 $O(\log_B n)$ 层。因此，可得以下定理：

定理14.3 在查找或更新操作中，具有 n 个数据项的B树执行 $O(\log_B n)$ 次磁盘传输，其中 B 是一个块中能够存放的数据项数。

每个内部结点至少有 $\lceil d/2 \rceil$ 个子结点的要求蕴涵着支持B树所用的每个磁盘块至少是半满的。在一棵B树中，分析和实验研究的结果表明，块平均利用率是67%，这相当不错。

653

14.1.3 外存排序

除了需要在外存中实现的数据结构（如字典）之外，还有许多算法必须操作在输入集上，由于数据集太大，不能完全装入内存。在这种情况下，目标是利用尽可能少的块传输，来解决算法上的问题。这种外存算法最经典的领域是排序问题。

1. 外存排序下界

正如上面讨论的那样，一个内存算法的性能与一个外存算法的性能存在巨大差异。例如，外存基数排序算法的性能不佳，而内存基数排序算法的性能良好。然而其他算法（如归并排序）的性能在内存和外存中均适中。传统归并排序算法进行的块传输次数为 $O((n/B) \log_2 n)$ ，其中 B 是磁盘块的大小。这个结果要比外存基数排序进行的 $O(n)$ 次传输好得多。然而，它不是排序问题所得结果中最好的结果。事实上，我们可以证明以下下界，其证明超出了本书的范围。

定理14.4 对存储在外存中的 n 个数据项进行排序，需要

$$\Omega\left(\frac{n}{B} \cdot \frac{\log(n/B)}{\log(M/B)}\right)$$

次块传输，其中 M 是内存大小。

比率 M/B 是恰好适合内存的外存中的块数。因此，这个定理是说排序问题能够达到的最好性能等价于至少要对输入集（需要 $\Theta(n/B)$ 次传输）扫描对数次，对数的底是适合内存中的块数。我们不会形式化证明这个定理，但会显示如何设计一个外存排序算法，其运行时间在这个下界的一个常数因子之内。

2. 多路归并排序

在外存中对 n 个对象的集合 S 进行排序的有效方式，相当于对熟悉的归并排序算法进行一些适合外存的简单改变。这种变化背后的主要思想是每次递归归并多个有序的表，从而减少递归的层数。确切地讲，对这个多路归并排序（multi-way merge-sort）方法的高级描述是，把 S 划分成大致相等的 d 个子集 S_1, S_2, \dots, S_d ，递归对每个子集 S_i 进行排序，然后同时把所有 d 个有序表归并为 S 的有序表。如果执行归并过程能够只利用 $O(n/B)$ 次磁盘传输，那么，对于足够大的 n 值，这个算法进行传输的总次数满足以下递归方程：

$$t(n) = d \cdot t(n/d) + cn/B$$

其中 $c \geq 1$ 为常数。当 $n \leq B$ 时，停止递归，因为此时可以执行单一块传输，把所有对象装入内存，然后用有效内存算法对这个数据集排序。因此， $t(n)$ 的停止准则是

$$t(n) = 1, \text{ 如果 } n/B \leq 1$$

这蕴涵着 $t(n)$ 的闭型解为 $O((n/B) \log_d(n/B))$ ，它是

$$O((n/B) \log(n/B) / \log d)$$

因此，如果可以选择 d 为 $\Theta(M/B)$ ，那么这个多路归并排序算法最坏情况下进行的块传输次数将是定理14.4给出的下界的常数因子。选择

$$d = (1/2) M/B$$

因而，这个算法剩下要解决的唯一问题是，如何只利用 $O(n/B)$ 次块传输进行 d -路归并。

通过运行“锦标赛树”执行 d -路归并。设 T 是 d 个外部结点的完全二叉树，保持 T 完全在内存中。使 T 的每个外部结点 i 关联一个不同的有序表 S_i 。通过读入每个外部结点 i 即 S_i 中的第一个对象，来初始化 T 。这起到把每个有序表 S_i 中的第一个块装入内存中的作用。对于两个外部结点的每个内部父结点 v ，然后比较存储在 v 的子结点处的对象，并把这两者中较小的与 v 关联。然后在 T 的向上一层重复这个比较测试，之后再上一层，依此类推。当到达 T 的根 r 时， r 关联所有表中最小的对象。这就完成了 d -路归并的初始化过程（见图14-4）。

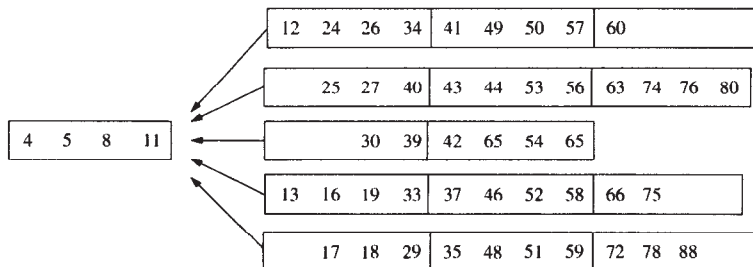


图14-4 d -路归并。 $B=4$ 时的五路归并

在 d -路归并的常规步骤中,把关联 T 的根 r 的对象 o 移到正在构建的归并表 S' 的数组中,然后沿着从外部结点 i 到 o 的路径,从 T 向下探索。然后把表 S_i 中的下一个对象读入 i 。如果 o 不是它所在块中的最后一个元素,那么此下一个对象已在内存中。否则,读 S_i 中的下一个块,访问这个新对象(如果 S_i 现在为空,则给结点 i 关联一个关键字为 $+\infty$ 的伪对象)。然后,对于从 i 到 T 的根中的每个内部结点重复最小值计算。再次可得完全二叉树 T 。接着从 T 的根到归并表 S' 重复这个移动对象的过程,并重建 T 直到 T 中对象为空。归并中的每一步所需时间为 $O(\log d)$;因此, d -路归并的内部时间为 $O(n \log d)$ 。归并中进行的传输次数为 $O(n/B)$,因为按序扫描每个表 S_i 一次,并写出归并表 S' 一次。由此可得:

定理14.5 给定基于数组的存储在外存中的 n 个元素的序列 S ,对 S 排序所用的传输次数为 $O((n/B)\log(n/B)/\log(M/B))$,内部CPU时间为 $O(n \log n)$, M 是内存大小, B 是块大小。

3. 达到“几乎”独立于机器

利用B树和外存排序算法可以大大减少块传输的次数。产生这种可能减少的最重要的一点是知道磁盘块(或高速缓存管线)大小 B 的值。这个信息当然是独立于机器的,但它是为数不多的真正独立于机器所需的信息之一,利用其中的一种信息,就能够把关键字连续存储在数组中。

根据我们对B树和外存排序的描述,可能认为还需要低级访问外部设备驱动器,但为了达到所声称结果的一个常数因子,这不是严格需要的。尤其是,除了知道块大小之外,唯一需要知道的其他事情是把关键字组成的大数组划分成连续单元组成的块。这样可使我们在B树中实现“块”,并且外存排序算法把它们作为各自大小为 B 的数组,我们称之为伪块(pseudo-block)。如果按照自然方式对数组分块,任何这样的伪块至多被分配到两个实际的块中。因此,即使我们准备依靠操作系统进行块的替换(例如,利用FIFO、LRU或稍后在14.3节中讨论的Marker策略),可以确信访问任何伪块至多需要2次真正的块传输,即为 $O(1)$ 。于是通过利用伪块而不是真正的块,可以实现字典ADT,使查找和更新操作只利用 $O(\log_B n)$ 次块传输。于是,我们可以设计外存数据结构和算法,无需由操作系统完全控制存储器的层次结构。

656

14.2 并行算法

在这一节里,讨论并行算法和某些基本的并行算法技术,包括简单并行分治法、串行子集、Brent定理、双递归和并行前缀,以及并行归并和排序。所有这些技术已证明对于设计大量有效的并行算法是有用的。最后把这些技术应用于找出凸多边形直径的问题。

14.2.1 并行计算模型

扩展RAM模型使之适合于多处理器,引出了称为并行RAM或PRAM的并行模型。这是一种同步并行模型,所有处理器共享存储空间。由于其概念简单,这个模型几乎是所有计算机以这样或那样方式仿效的模型。PRAM模型还非常适合于找出问题中可能存在的固有并行性。最后,PRAM模型似乎理想地适合于一般范型的模型,可用于开发有效的并行算法。

1. 并行效率

在串行设置中,如果一个算法的运行时间至多为 $O(n^k)$,其中 k 为常数,即它的运行时间是多项式时间,则称这个算法是好的。正如在第13章中描述的那样,如果存在求解一个问题的多项式时间算法,则认为这个问题是“易解的”。并行设置中的相应概念是说,如果一个算法的运行时间是 $O(\log^{k_1} n)$,利用 $O(n^{k_2})$ 个处理器,其中 k_1 和 k_2 是常数,即它利用多项式个处理器,运行时间为

多项式时间, 则称该算法是好的。类似地, 在复杂度理论中, 如果存在这种意义上的求解它的好算法, 则称这个问题属于NC类。

在串行设置中, 一旦说一个问题可在多项式时间内求解, 目标就转到找出求解这个问题的尽可能快的算法上。同样, 在并行设置中, 一旦知道一个问题是在NC中, 目标就转到找出求解这个问题的算法, 且使 $T(n) * P(n)$ 达到最小(渐近意义上), 其中 $T(n)$ 为算法的时间复杂度, $P(n)$ 是算法中使用的处理器数量。也就是说, 如果 $Seq(n)$ 表示求解某一问题的串行运行时间, 那么在渐近意义上希望使 $T(n) * P(n)$ 尽可能接近 $Seq(n)$ 。促使达到这个目标是由于如下事实: 即单处理器能够按照轮询方式, 每次执行一步, 模拟 $P(n)$ 个处理器的计算(见2.1.2节)。

657

如果 $T(n) * P(n)$ 匹配求解问题的串行下界, 则称一个并行算法是最优的(optimal)。技术上讲, 这个定义允许一个串行算法($P(n) = 1$)也可以是一个最优的并行算法。因此, 给定某些问题的 $T(n) * P(n)$ 接近于 $Seq(n)$, 第二个目标是最小化运行时间 $T(n)$ 。这两个目标的动机是说, 任何具有(比如说) k (一个常数)个处理器的现有机器都可以模拟在这种意义上有效的算法, 达到最大加速量。通过使这 k 个处理器每个执行 $P(n)/k$ 的工作, k 个处理器可以用 $P(n)/k$ 时间模拟 $P(n)$ 个处理器中的每个时间步。这种方法导致算法的运行时间为 $O(T(n)P(n)/k + T(n))$ 。此外, 求解某一问题所需串行运行时间的下界直接变成 $T(n) * P(n)$ 的下界(因为一台串行机器可以用 $T(n) * P(n)$ 的时间界限模拟一个PRAM算法)。

2. PRAM模型的版本

实质上有三种不同的PRAM模型, 它们之间的差异在于如何解决内存冲突(回忆在所有PRAM模型中, 处理器同步工作)。最受限的版本是互斥读、互斥写(或EREW)PRAM, 在这个模型中, 不允许同时读或同时写。如果允许多个处理器同时读取同一内存单元, 但仍然限制并发写, 那么得到并发读、互斥写(或CREW)PRAM。最后, 如果允许多个处理器同时写同一内存单元, 那么得到最强大的模型, 即并发读、并发写(或CRCW)PRAM。有多个解决写冲突的方法, 但基于以下其中一条规则, 基于定义的模型有两种最基本的方法:

- 限制所有处理器向同一位置写入同一个值。
- 当 $k > 1$ 个处理器同时写入同一位置时, 只允许其中一个处理器写成功(任意)。

作为CRCW PRAM模型算法的一个例子, 假定给定一个布尔数组 A , 要求计算 A 中所有位的公共OR。我们可以初始化输出位 o 为0, 并为 A 中的每一位分配一个处理器。然后, 使分配给一位的每个的处理器向 o 中写入值1。基于CRCW PRAM模型, 这个简单的算法利用 n 个处理器计算 n 位中的OR的时间为 $O(1)$ (利用上述规则之一解决写冲突)。

658

这个简单的例子显示了CRCW PRAM模型的力量, 但许多研究人员认为这个模型是不现实的。因此, 本节中描述的所有其他算法将用于CREW PRAM模型和EREW PRAM模型。

14.2.2 简单并行分治法

并行分治技术与经典的串行分治技术完全类似。方法如下。给定一个问题, 我们把问题划分成在结构上与原问题类似的许多更小的子问题, 然后递归并行求解每个子问题。为了得到有效的算法, 一旦并行递归调用返回时, 必须能够快速并行归并子问题的解。

作为利用这种技术的一个简单例子, 考虑计算 n 个整数之和的问题(同样的技术可用于任何相关操作)。假设输入是包含 n 个元素的数组, 并且已经驻留在存储器中。为简单起见, 假设 n 为2的幂。把数组划分成大小都为 $n/2$ 的子数组, 并行求每个子数组中整数的和。并行递归调用返回之后, 把每个子数组计算的和相加, 得到所有 n 个整数的和。由于这一步所需时间为 $O(1)$, 时间复杂度 $T(n)$ 满足递归方程 $T(n) = T(n/2) + c$, 这蕴涵着 $T(n) = O(\log n)$ 。这个计算所需的处理器数 $P(n)$

可以表示为递归方程 $P(n) = \max\{2P(n/2), 1\}$, 它的解为 $P(n) = O(n)$ 。注意在这种情况下, $T(n) * P(n) = O(n \log n)$ 。它距最优相差一个 $\log n$ 因子。在下一节里, 讨论两个用于减少处理器数量的方法, 这两个方法使 $T(n) * P(n)$ 是最优的, 即为 $O(n)$ 。

14.2.3 串行子集和 Brent 定理

串行子集 (sequential subset) 技术和 Brent 定理 (Brent's theorem) 都涉及通过现有的算法构造一个有效的并行算法有关, 这些现有的算法自身可能效率不高。我们从串行子集技术开始讨论。这项技术隐含的主要思想是想要对求解问题的小子集进行有限数量的串行预处理, 然后应用并行算法完成这个问题的求解。在某些情况下, 并行算法完成之后, 每个子集可能还有一些串行后处理需要完成。我们说明如何将这项技术应用于 n 个整数的求和问题。首先, 可以把数组划分成 $n/\log n$ 个子数组, 每个子数组的大小为 $\log n$ 。为每个子数组分配一个处理器, 每个处理器串行求元素的和。然后, 可以利用上述的并行分治过程只对 $n/\log n$ 个部分和进行求和。预处理步骤利用 $O(n/\log n)$ 个处理器, 所需运行时间为 $O(\log n)$, 正如在并行分治步骤中所做的那样。因此, 这导致计算 n 个整数和的最优结果, 即使得 $T(n) * P(n) = O(n)$ 。

659

Brent 定理

Brent 定理也是减少求解特定问题所需处理器数量的一项技术。如果设计的算法在许多计算时间里使大量处理器处于空闲状态, 那么 Brent 定理可能有用。这个定理概括如下:

定理 14.6 对于任何运行 T 个时间步且由 N 个操作组成的同步并行算法, 可用 P 个处理器进行模拟, 所需时间为 $O(\lfloor N/P \rfloor + T)$ 。

证明 (概略证明) 设 N_i 是并行算法的第 i 步执行的操作数。 P 个处理器可以用 $O(\lceil N_i/P \rceil)$ 的时间模拟算法的第 i 步。因此, 总运行时间为 $O(\lfloor N/P \rfloor + T)$, 因为

$$\begin{aligned} \sum_{i=1}^T \lceil N_i/P \rceil &\leq \sum_{i=1}^T (\lfloor N_i/P \rfloor + 1) \\ &\leq \lfloor N/P \rfloor + T \end{aligned}$$

■

然而在把 Brent 定理应用于 PRAM 模型时, 必须首先满足两个条件。其一, 在第 i 步开始, 必须能够利用 P 个处理器在 $O(\lceil N_i/P \rceil)$ 时间内计算 N_i 。也就是说, 必须知道在第 i 步中实际有多少个操作要被执行。其二, 必须准确知道分配给每个处理器的工作量。也就是说, 在它执行模拟第 i 步时, 必须能够给每个处理器分配 $O(\lceil N_i/P \rceil)$ 个操作。

作为 Brent 定理的一个应用例子, 再次考虑求和问题。回忆上面提出的分治求和算法。尽管算法的运行时间为 $O(\log n)$ 且利用 $O(n)$ 个处理器, 但它执行的总操作数为仅 $O(n)$ 。因此, 通过应用 Brent 定理, 得到求解这个问题的另一种方法, 它利用 $O(n/\log n)$ 个处理器, 运行时间为 $O(\log n)$ 。在这种情况下 Brent 定理的两个条件易于满足, 因为第 i 步中执行的操作数恰好为第 i 减步1步中执行操作数的一半。更确切地讲, 模拟第一步需要 $\lceil \log n \rceil$ 的时间 (所有 n 个处理器处于活动状态), 模拟第二步需要 $\lceil \log n/2 \rceil$ 的时间, 模拟第三步需要 $\lceil \log n/4 \rceil$ 的时间, 依此类推。这个和显然为 $O(\log n)$, 这给出利用 $O(n/\log n)$ 个处理器, 运行时间为 $O(\log n)$ 的求 n 个整数之和的另一种方法。

660

14.2.4 递归倍增

递归倍增是一种技术, 可以认为是对分治范型的一种补充——分治法是一种自顶向下的技术, 而递归倍增是一种自底向上的技术。其主要思想是从小的子集开始, 反复成对组合它们, 直

到解决整个问题。我们用一个例子即表序 (list ranking) 问题描述递归倍增技术。在这个表序问题中, 给定一个链表, 表示驻留在内存中的指针数组。要求计算表中每个元素距表尾的距离。这里讨论这个问题的经典并行算法。其主要思想是把表中每个元素分配给一个处理器, 且在每个时间步, 使每个元素的指针指向其后继指针。随着每次迭代, “看似前面有元素” 的每个元素的距离都会加倍, 因此, 至多经过 $O(\log n)$ 次迭代, 每个元素都会指向表尾。更确切地讲, 设 $p(v)$ 表示来自 v 的指针, $tail$ 表示表尾所在的结点。此外, 每个结点 v 中存储标记 $r(v)$, 除了尾结点 (它的 r 标记为 0) 之外, 所有结点的标记都初始化为 1。这个标记最终存储 v 在表中的位序。在每个时间步中, 可以并行执行每个结点 v 的以下操作 (见图 14-5)。

```

if  $p(v) \neq tail$  then
     $r(v) := r(p(v)) + r(v);$       {排序步骤}
     $p(v) := p(p(v));$              {"倍增"步骤}
  
```

因此, 可得以下定理:

定理 14.7 对于 n 个结点的表 L , 在 CREW PRAM 模型中, 利用 $O(n)$ 处理器, 找出其中每个结点 v 到表尾的距离可在 $O(\log n)$ 时间内完成。

注意上述 $T(n) * P(n)$ 距最优差一个 $\log n$ 因子。

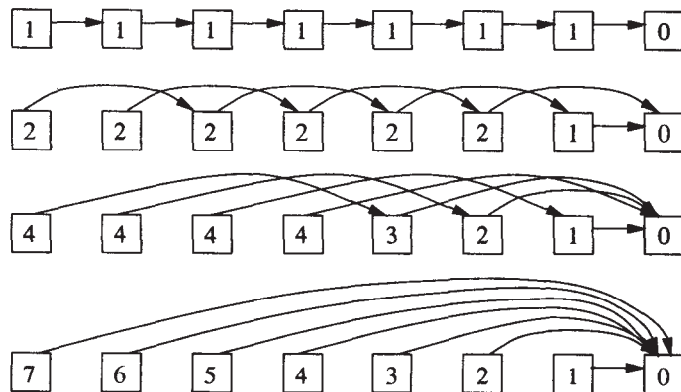


图 14-5 将递归倍增应用于表序问题

并行前缀

有一个相关的问题, 它是并行前缀求和问题, 也可用递归倍增技术求解。给定整数组成的表 $A = (a_1, a_2, \dots, a_n)$, 希望计算所有前缀和 $s_k = \sum_{i=1}^k a_i$ 。利用递归倍增技术, 可以容易地利用 $O(n)$ 个处理器用 $O(\log n)$ 时间解决这个问题, (见图 14-6)。这个方法与上述给出的求解表序问题的方法相同。

与表序问题不同的是, 我们可以容易地把串行子集技术相当直接地应用于并行前缀问题。在这种情况下, 可以把处理器数量减少到 $O(n/\log n)$, 而运行时间仍为 $O(\log n)$ 。这时得到最优乘积 $T(n) * P(n)$ 。求解方法如下。把数组 A 划分成 $n/\log n$ 个子数组, 每个子数组大小为 $\log n$, 每个子数组利用一个处理器串行求和。那么利用这些部分和作为元素, 如前所述进行前缀计算。完成这些元素的前缀计算之后, 通过 $n/\log n$ 个子数组中的每一个简单 “回溯”, 为每个子数组分配一个处理器, 计算所有部分和 (见图 14-7)。

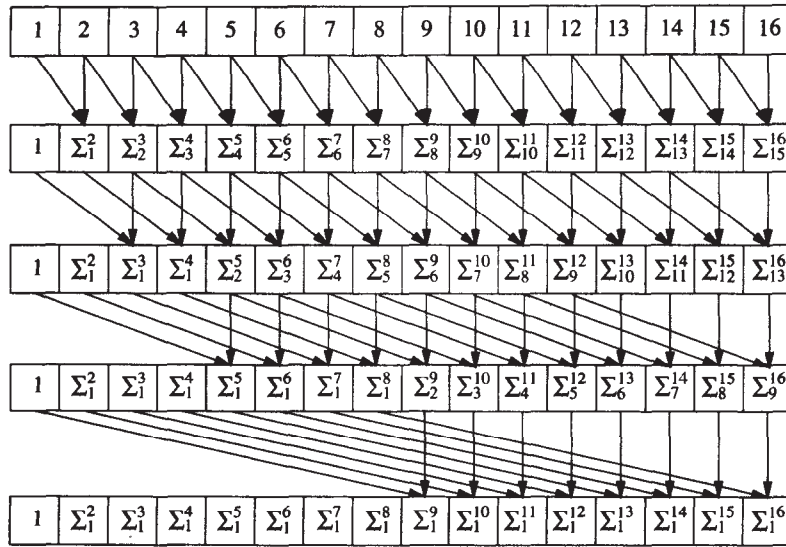


图14-6 将递归倍增应用于并行前缀问题

662

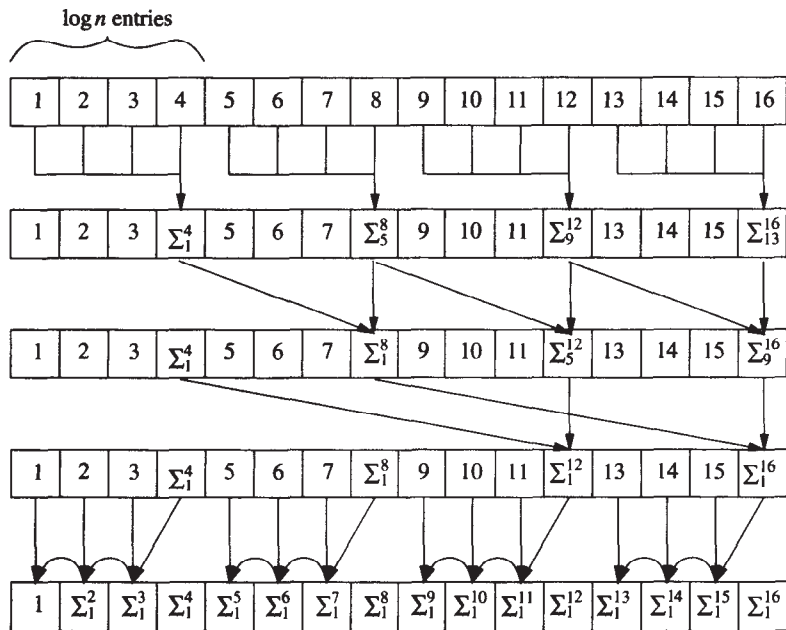


图14-7 将串行子集应用于并行前缀问题

在许多其他的表操纵问题中，并行前缀问题作为子问题出现。我们用两个例子说明它。

第一，考虑下列数组压缩问题：给定 n 个元素的数组 A ，其中每个元素标记为“marked”或“unmarked”，构造数组 B ，按照在 A 中出现的次序列出 A 中标记过的元素（即 B 是 A 的标记序列）。这个方法把每个标记过的元素关联一个值1，而把未标记过的元素关联一个值0。通过进行并行前缀计算，可以确定 A 中每个标记过元素在压缩表 B 中的位序。可以利用 $O(n)$ 个处理器，在 $O(\log n)$ 时间完成它。

第二，考虑数组分裂（array splitting）问题：给定 n 个元素组成的数组 A ，把 A 分解为子数组

A_1, A_2, \dots, A_m , 使得每个 A_i 由相同整数重复组成, 它与 A_{i-1} 和 A_{i+1} 中重复的整数不同 (即 $k \in A_i$, 蕴涵着 $k \notin A_{i-1}$ 和 $k \notin A_{i+1}$)。可以把 A 中每个元素关联值 1, 只要前面所有元素都相等, 就对每个元素执行并行前缀算法的组合步骤。然后可以执行另一个并行前缀操作, 确定 a 属于哪个数组 A_i (即确定 i 的值)。

663

14.2.5 并行归并和排序

归并两个有序表或对 n 个数进行排序的计算, 在串行模型下可以容易地有效进行, 但要有效并行进行这些计算却并不那么简单。对于这两个问题, 有相当多的工作要做, 用最有效的归并算法能够利用 $n / \log n$ 个处理器在 $O(\log n)$ 时间内归并两个有序表。最有效的排序算法能够利用 n 个处理器在 $O(\log n)$ 时间内对 n 个数进行排序。此外, 有些算法甚至可以运行在 EREW PRAM 模型中, 同时保持相同的渐近界限。

不幸的是, 最有效的并行归并算法和排序算法太棘手, 这里不能详细描述。我们描述归并两个有 n 个不同元素的有序表的简单算法, 所需时间为 $O(\log n)$, 它基于 CREW PRAM 模型, 利用 n 个处理器。对于任何元素 x , 定义 x 在 A 或 B 中的位序 (rank) 为表中小于或等于 x 的元素个数。算法找出 A 中每个数据项在 B 中的位序, 以及 B 中每个数据项在 A 中的位序。尤其是, 为 A 和 B 中的每个数据项分配给一个处理器, 执行二分搜索, 找出那个数据项在另一个表中的位置。这需要 $O(\log n)$ 的时间, 因为所有二分搜索可以并行执行。一旦被分配给 $A \cup B$ 中的元素 x 的处理器知道 x 在 A 中的位序 i 及其在 B 中的位序 j , 那么那个处理器可以快速地把 x 写入到归并输出数组中的 $i + j$ 处。因此, 可以利用 n 个处理器在 $O(\log n)$ 时间内归并 A 和 B 。

一旦知道如何快速并行归并两个有序表, 可以利用这个算法作为并行归并排序算法中的子例程。确切地讲, 对集合 S 排序, 把 S 划分为两个大小相等的集合 S_1 和 S_2 , 并行递归地对每一部分进行排序。一旦完成对 S_1 和 S_2 的排序, 那么利用并行归并算法把它们归并成为有序表。所需的处理器总数为 n , 因为分配给每个子问题的处理器数量为其自身的大小。同样, 算法的运行时间 $T(n)$ 可由递归方程 $T(n) = T(n/2) + b \log n$ 表征, 其中 b 为常数。这蕴涵着 $T(n)$ 是 $O(\log^2 n)$ 。因此, 可得以下定理:

定理 14.8 给定 n 个数据项的集合 S , 可在 CREW PRAM 模型中, 利用 n 个处理器用 $O(\log^2 n)$ 时间对 S 排序。

注意上述算法是简单并行分治模式的一个例子。在习题 C-14.23 中探讨如何用 $O(\log n)$ 时间归并两个有序数组, 利用 $O(n / \log n)$ 个处理器, 这会产生工作最优的并行排序算法。如上所述, 实际上可通过更复杂的算法, 用 $O(\log n)$ 时间对 n 个数据项排序, 该算法基于 EREW PRAM 模型, 利用 n 个处理器。尽管这里没有描述这个算法, 但知道这个事实对于构建其他有效的并行算法是有用的。下一节给出这个应用的一个例子, 把归并算法和排序算法 (以及上面介绍的其他技术) 应用于几何问题。

664

14.2.6 找出凸多边形的直径

考虑以下问题: 给定一个凸多边形 P , 找出 P 上一对相距最远的点。这如同计算 P 的直径 (diameter), 即 P 上一对最远的点之间的距离。容易看出对于任意 P , 存在一对最远的点, 它们都是 P 的顶点。因此, 可以容易地利用 $O(n^2)$ 个处理器在 $O(\log n)$ 时间内求解这个问题 (利用简单分治技术计算所有 $O(n^2)$ 个顶点之间的最大距离)。然而, 利用问题中已有的几何结构以及上述一些技术可以做得更好。在这一节里, 提出一个 DIAMETER 算法求解找出直径的问题, 该算法所需时间

为 $O(\log n)$ ，利用 $O(n/\log n)$ 个处理器。它是一个最优算法。

注意任意一对最远的点 p 和 q 必定是 P 中的对映(antipodal)顶点。点 p 和 q 是对映的，如果有两条与多边形 P 相切的平行线 L_1 和 L_2 ，使 L_1 包含 p 且 L_2 包含 q 。这显然使必须考虑的点对数从 $O(n^2)$ 降至 $O(n)$ ，但仍有一个问题，就是如何并行有效地枚举出顶点的所有对映对。沿着 P 上顶点的循环次序确定了每条边方向。

把每条边看作一个向量，把这个边向量集合转换成原点。任何通过这个向量图原点的直线都会横截两个扇区，这两个扇区与对映顶点对应(见图14-8)。

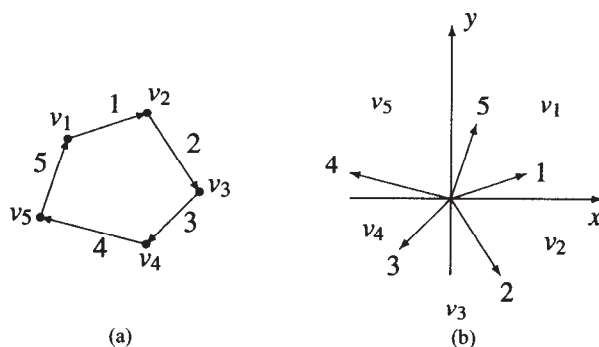


图14-8 把边看作向量，并把这些边转换成原点。注意多边形(a)中的顶点对应于向量图(b)中的扇区

665

因为我们在 $O(\log n)$ 时间内找出所有对映顶点，不能利用旋转包含原点的直线通过向量集的方法。也不能为每个扇区(对应某个顶点 v)分配一个处理器，然后用二分搜索枚举所有与 v 对映的其他顶点，因为对于任何 v ，这样的顶点可能有 $O(n)$ 个，此外，我们只能支配 $O(n/\log n)$ 个处理器。作为替代，用 x 轴将向量集划分成两个集合，把 x 轴下方的所有向量旋转 180° ，并利用并行过程枚举出所有对映顶点。然后，通过取 $O(n)$ 对中的最大值，找出 P 中一对最远的顶点(见算法14-1)。

算法14-1 找出凸多边形 P 上的两个最远点的算法

算法 DIAMETER(P):

输入: 凸多边形 $P = (v_1, v_2, \dots, v_n)$

输出: P 中两个最远的点 p 和 q

1. 利用 P 中顶点的循环次序，确定每条边的方向，把这些边看作向量，把这个边向量集转换成原点。
2. 标记向量图中的向量，将 x 轴上方的向量标为“红色”，将 x 轴下方的向量标为“蓝色”。
3. 将每个蓝色向量旋转 180° 。
{红色向量和蓝色向量现在按角度进行了相似的排序。}
4. 利用快速并行归并算法归并这两个有序表。
5. **for** 每个向量 v **do**
 利用有序表计算两个相反颜色的顶点，使其在归并表中的位置是：一个在 v 之前，另一个在 v 之后。
 确定每个这样的对 (a, b) 作为对映对。
6. 找出对映顶点之间所有距离中的最大值，以找出 P 中最远的顶点对。

定理14.9 算法DIAMETER正确地找出 n 个顶点的凸多边形 P 上一对最远的点，所用时间为 $O(\log n)$ ，该算法基于CREW PRAM模型，利用 $O(n/\log n)$ 个处理器。

证明 算法DIAMETER的正确性由以下事实而得：即把x轴以下的向量旋转180°，把以前相反的（即两个扇区被同一条过原点的直线所截）所有扇区现在重叠。我们转向复杂度的界限。注意如果要用 $O(n)$ 个处理器，则第1、2、3和5步可在 $O(1)$ 时间内求解，因为在这些步骤的每一步中，正在为 $O(n)$ 个对象中的每一个对象做 $O(1)$ 的工作。因此，可以利用 $O(n/\log n)$ 个处理器用 $O(\log n)$ 的时间执行这些步骤中的每一步。观察可知，第4步和第6步可利用 $O(n/\log n)$ 个处理器用 $O(\log n)$ 时间完成（见习题C-14.23）。 ■

666

14.3 在线算法

在线算法对应于一系列服务请求，每个请求关联代价（cost）。例如，网页置换策略把页面维持在高速缓存中，对于一系列访问请求，满足如果请求页面在高速缓存中，则页面请求代价为0；如果请求页面不在高速缓存中，则页面请求代价为1。在在线（online）设置中，算法必须完全完成对服务请求的响应，才能接收序列中的下一个请求。如果算法预先给出服务请求的整个序列，则称它是离线（offline）算法。为了分析一个在线算法，我们常常利用竞争性分析，把某个在线算法 A 与一个最优离线算法 OPT 进行比较。给定某个服务请求序列 $P = (p_1, p_2, \dots, p_n)$ ，设 $cost(A, P)$ 表示 A 在 P 上的代价， $cost(OPT, P)$ 表示最优算法在 P 上的代价。对于 P ，如果

$$cost(A, P) \leq c \cdot cost(OPT, P) + b$$

其中 $b \geq 0$ 为常数，则称算法 A 是 c 可竞争的（ c -competitive）。如果对于每个序列 P ， A 是 c 可竞争的，那么简单地称 A 是 c 可竞争的，且称 c 为 A 的竞争率（competitive ratio）。如果 $b = 0$ ，那么称算法 A 有严格（strict）的竞争率 c 。

租赁人难题

一个最好用故事解释的众所周知的在线问题是租赁人难题（Renter's dilemma）。Alice决定利用Streamin Meemees收听某些歌曲来试验音乐流服务。每次Alice收听一首歌，要花费她 x 美元“租赁”那首歌，因为如果不再次付费，她的软件并不允许重放。假定花费 y 美元购买Streamin Meemees新专辑中的所有歌曲。为了讲故事的需要，称 y 比 x 大10倍，即 $y = 10x$ 。Alice的难题是，决定她何时应该买专辑，而不是一次一首地收听流式歌曲。例如，如果她在收听任意流式歌曲之前购买，然后决定她并不喜欢，那么她已经花费的钱比应该花的钱多10倍。但是如果她收听了许多次流式歌曲，且从未买过专辑，那么她甚至可能花费比应该花费的钱多10倍还多。事实上，如果她收听流式歌曲 n 次，那么这个总是“租赁”的策略使她花费的钱是应该花费的钱的 $n/10$ 倍，即首次购买策略最坏情况下的竞争率为10，总是租赁策略最坏情况下的竞争率为 $n/10$ 。这两个选择都不好。

幸运的是，Alice有一个竞争率为2的策略。也就是说，她可以租赁10次，然后买专辑。最坏情况的方案是她从来不听她刚买的专辑。因而，她花费了 $10x + y = 2y$ 美元，而本应花费 y ；因此，这个策略的竞争率为2。事实上，不论 y 比 x 大多少，如果Alice首先租赁 y/x 次，然后再购买，她的竞争率就为2。

667

14.3.1 高速缓存算法

有几种网络应用必须处理出现在网页中的再次访问信息。这些再次访问已经展示了在时间和空间上的引用局部性，为了充分利用这些引用的局部性，通常把网页的副本存储在高速缓存中比较好。因此当再有请求时，可以快速检索这些网页。特别是，假定有一个高速缓存，它有 m 个“单元”可以容纳网页。假设可把网页放在这些高速缓存中的任何单元中。称它为完全关联的（fully associative）高速缓存。

当一个浏览器执行时,请求不同的网页。每当浏览器请求一个网页 l 时,浏览器确定(利用快速测试) l 是否未发生变化且当前包含在高速缓存中。如果 l 当前在高速缓存中,那么浏览器利用高速缓存中的副本满足这个请求。但是,如果 l 不在高速缓存中,则在因特网上请求该页面 l ,并传回到高速缓存中。如果高速缓存中的 m 个单元之一可用,那么浏览器把 l 分配给其中一个空单元。但如果这 m 个高速缓存单元都被占据,那么在把 l 放入高速缓存之前,计算机必须确定将以前查看过的网页中哪一个网页从高速缓存中替换出。当然,可用多种策略确定替换出哪个页面。以下给出一些著名的页面替换策略(见图14-9):

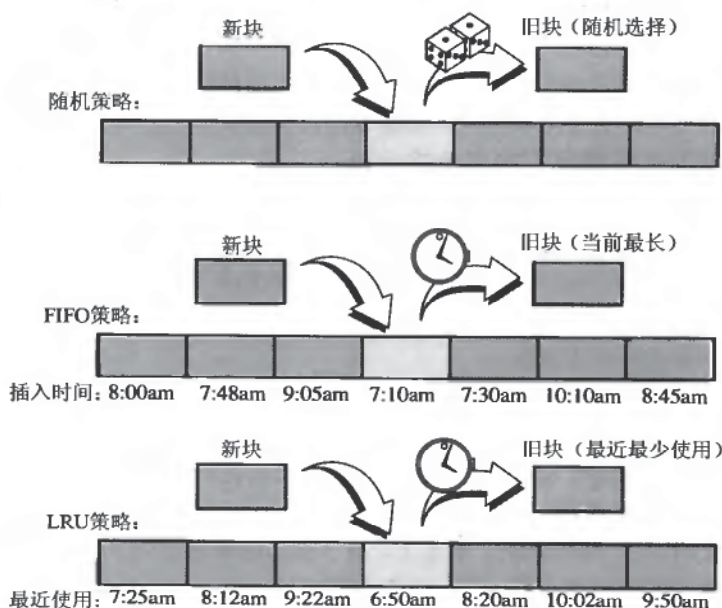


图14-9 随机、FIFO和LRU页面替换策略

- 先进先出 (first in, first out, FIFO) 策略: 替换出高速缓存中存放时间最长的一个页面, 即替换出过去传输到高速缓存中的时间最久的页面。
- 最近最少使用 (least recently used, LRU) 策略: 替换出过去最长时间未被请求的页面。此外, 可以考虑一种简单的纯粹随机策略。
- 随机 (random) 策略: 随机选择一个页面, 替换出高速缓存。

随机策略是最容易实现的策略。因为它只要求随机数或伪随机数生成器。每次页面替换发生时, 实现这个策略涉及的开销为 $O(1)$ 个额外的工作量。此外, 对于每次页面请求, 除了确定一个页面是否在高速缓存中的开销之外, 没有其他额外的开销。然而, 这个策略没有尝试利用用户浏览器展示的时间局部性和空间局部性。

FIFO策略相当简单, 易于实现, 因为它只需要队列 Q 把指向页面的引用存储在高速缓存中。当浏览器引用页面时, 把页面插入队列 Q , 然后, 把页面放入高速缓存中。当需要替换出页面时, 计算机简单地执行 Q 上的出队操作, 确定替换出哪个页面。因此, 每次页面替换发生时, 这个策略也要求 $O(1)$ 个额外的工作量。同样, 对于页面请求, FIFO策略不需额外的开销。此外, 它试图利用某种时间局部性。

LRU策略比FIFO策略前进了一步, 它假设已在高速缓存中存放时间最长的页面在未来被请求的可能性最小。因为LRU策略明确地尽可能利用时间局部性, 其方式是总是替换出最近最少使用的页面。从策略角度讲, 这是一种优秀的方法, 但从实现角度讲, 这是一种昂贵的方法, 即它优化时间局部性和空间局部性的方式相当昂贵。实现LRU策略要求利用优先队列 Q , 支持对现有页面的查找, 例如, 利用特殊指针或“定位器”。如果用基于链表的有序队列实现 Q , 那么每次请求页面和替换页面的开销为 $O(1)$ 。但当在 Q 中插入或更新一个页面时, 页面被赋予 Q 中的一个最大关键字, 并被放在表尾, 完成这些操作的时间为 $O(1)$ 。即使利用上述实现, LRU策略有常数时间的开销, 但是, 从实用角度来看, 涉及优先队列 Q 的额外时间开销和额外空间开销的常数因子使得这个策略吸引力不大。

因为这些不同的页面替换策略在实现难度以及利用局部性上的侧重有所不同, 通过对这些方法进行某种竞争性分析, 会得出哪一种策略是最好的 (如果存在最好的策略)。

1. FIFO和LRU最坏情况下的竞争性分析

从最坏情况来看, FIFO和LRU策略相当缺乏竞争行为。例如, 假定有一个高速缓存, 包含 m 个页面。考虑程序用FIFO和LRU方法进行页面替换, 该程序中有一个循环, 按照循环次序反复请求 $m+1$ 个页面。对于这样的页面请求序列, FIFO和LRU策略性能都不佳, 因为对于每个页面请求, 都要进行页面替换。因此, 从最坏情况分析来看, 这些策略几乎是最坏的, 我们能够想象——每次请求页面时, 都要进行页面替换。

然而最坏情况分析有点太悲观, 因为对于一个糟糕的页面请求序列, 它关注的是每个协议的行为。一个理想的分析应该针对所有可能的页面请求序列, 对这些方法进行比较。当然, 不可能穷尽所有的请求序列, 但是有大量对真实程序产生的页面请求所做的试验模拟。这些实验主要关注的是随机策略、FIFO策略和LRU策略。基于这些实验比较, 策略从最好到最坏的次序是: (1)LRU, (2) FIFO和 (3) 随机策略。事实上, 对于典型的请求序列, LRU要比其他策略好很多, 但在最坏情况下它的性能仍然不好。如以下定理所表明的那样。

定理14.10 对于具有 m 个页面的高速缓存的FIFO和LRU页面替换策略, 其竞争率至少为 m 。

证明 由上述观察可知, 存在页面请求序列 $P = (p_1, p_2, \dots, p_n)$, 使得对于每次页面请求——循环中的 $m+1$ 次请求, FIFO和LRU都会执行页面替换。把这个性能与页面替换问题的最优离线算法 OPT 进行比较, 这个最优离线算法从高速缓存中替换出未来被请求的时间最久远的页面。当预先给出整个序列 P 时, 这个策略只能在离线情况下实现, 除非算法是“预言性的”。当应用到循环序列时, OPT 策略对于每 m 次请求 (因为它每次替换出最近被引用的页面, 因为这个页面在未来被引用的时间最久远) 执行一次页面替换。因此, 对于序列 P , FIFO和LRU都是 c 可竞争的, 其中

$$c = \frac{n}{n/m} = m$$

观察可见, 如果 P 的任一部分 $P' = (p_i, p_{i+1}, \dots, p_j)$ 请求 m 个不同的页面 (p_{i-1} 和 p_{j+1} 不在其中), 那么甚至最优的算法也必须替换出一个页面。此外, FIFO和LRU策略从 P' 中替换出的最多页面数为 m , 每次替换出一个在 p_i 之前引用的页面。于是, FIFO和LRU竞争率为 m , 且这可能是这些策略在最坏情况下可能最好的竞争率。 ■

2. 随机化Marker算法

与“预言性的”最优算法相比, 即使是确定性的FIFO和LRU策略, 它们在最坏情况下竞争率仍然可能不佳, 我们可以看到试图模拟LRU的随机策略有好的竞争率。确切地讲, 我们研究试

图仿效LRU策略的随机策略的竞争率。从策略的角度来看,称这个策略为Marker策略 (Marker strategy), 它仿效确定性LRU策略最好的方面, 同时利用随机化避免对于LRU策略非常糟糕的最坏情况。Marker策略如下:

- Marker策略: 把高速缓存中的每个页面关联一个布尔变量 “marked”, 起初设置每个高速缓存中的页面为 “false”。如果浏览器请求一个已在高速缓存中的页面, 则设置这个页面的标记变量为 “true”。否则, 如果浏览器请求的一个页面不在高速缓存中, 则将标记变量为 “false” 的一个随机页面替换出, 用一个新页面替换它, 并直接设置这个新页面的标记变量为 “true”。如果高速缓存中所有页面的标记变量都设置为 “true”, 那么将它们都重新设置为 “false” (见图14-10)。

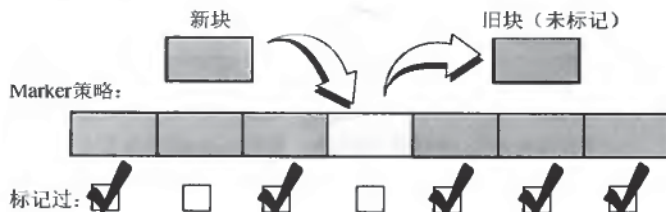


图14-10 Marker页面替换策略

3. 随机化在线算法的竞争性分析

有了上述策略定义, 我们现在想要进行Marker策略的竞争性分析。但是, 在进行分析之前, 首先必须定义随机化在线算法的竞争率。因为一个随机化算法 A (如Marker策略) 可能具有不同的运行结果, 这取决于所执行的随机选择, 对于请求序列 P , 如果

$$E(\text{cost}(A, P)) \leq c \cdot \text{cost}(\text{OPT}, P) + b$$

则定义这样的算法是 c 可竞争的, 其中 $b \geq 0$ 为常数, $E(\text{cost}(A, P))$ 表示算法 A 在序列 P 上的期望代价 (算法 A 取所有可能的随机选择的期望值)。如果对于每个序列 P , 算法 A 是 c 可竞争的, 那么可以简单地说 A 是 c 可竞争的, 并且称 c 是 A 的竞争率。

671

定理14.11 对于具有 m 个页面的高速缓存, Marker页面策略的竞争率为 $2\log m$ 。

证明 设 $P = (p_1, p_2, \dots, p_n)$ 是足够长的页面请求序列。Marker策略蕴涵着把 P 中的请求划分成轮 (round)。每一轮开始时, 高速缓存中所有页面的标记都为 “false”。一轮结束时, 高速缓存中所有页面的标记都为 “true” (下一个请求开始下一轮, 因为策略这时把每个这样的标记重新设置为 “false”)。考虑 P 中的第 i 轮, 如果在第 i 轮开始时请求的页面不在Marker策略的高速缓存中, 则称第 i 轮中请求的页面为最新的 (fresh)。同时, 称在Marker的高速缓存中标记为 “false” 的页面为陈旧的 (stale)。因此, 在第 i 轮开始时, Marker策略的高速缓存中的所有页面都是陈旧的。设 m_i 表示第 i 轮中引用的最新页面数, b_i 表示第 i 轮开始时存在于OPT算法的高速缓存中且此时不在Marker策略的高速缓存中的页面数。因为Marker策略对于 m_i 个请求中的每一个必须进行页面替换, 算法OPT在第 i 轮至少必须进行 $m_i - b_i$ 次页面替换 (见图14-11)。此外, 因为在第 i 轮结束时, Marker策略的高速缓存中的每个页面在第 i 轮中被请求, 算法OPT在第 i 轮中至少必须执行 b_{i+1} 次页面替换。因此, 第 i 轮中算法OPT至少必须执行

$$\max\{m_i - b_i, b_{i+1}\} \geq \frac{m_i - b_i + b_{i+1}}{2}$$

次页面替换。对 P 中的所有 k 轮求和, 那么可见算法OPT至少必须执行的页面替换次数如下:

$$L = \sum_{i=1}^k \frac{m_i - b_i + b_{i+1}}{2} = (b_{k+1} - b_1)/2 + \frac{1}{2} \sum_{i=1}^k m_i$$

第 i 轮中引用的 m_i 个最新块



图14-11 第*i*轮开始时, Marker的高速缓存状态和OPT的高速缓存状态

下面，我们考虑Marker策略执行的页面替换次数。

观察可见第 i 轮中Marker策略至少必须执行 m_i 次页面替换。然而实际上执行的替换次数可能还要多, 如果它替换出陈旧页面, 然后又在那一轮中请求这些页面。因此, Marker策略执行的页面替换的期望次数为 $m_i + n_i$, 其中 n_i 是第 i 轮中从高速缓存中被替换出而又在那一轮中被请求的陈旧页面的期望数。 n_i 值等于对于第 i 轮中引用的所有陈旧页面, 当引用时这些页面不在高速缓存中的概率之和。在第 i 轮陈旧页面 v 被引用的时刻, v 在高速缓存之外的概率至多为 f/g , 其中 f 是在页面 v 之前被引用的页面数, g 是还未被引用的陈旧页面数。这是因为每次引用最新页面都要随机替换出某些未标记的陈旧页面。如果所有对最新页面的 m_i 个请求发生在对陈旧页面的请求之后, 此时Marker策略的代价是最大的。因此, 假设从最坏情况下考虑, 第 i 轮中替换出被引用的陈旧页面的期望界限如下:

$$n_i \leq \frac{m_i}{m} + \frac{m_i}{m-1} + \frac{m_i}{m-2} + \dots + \frac{m_i}{m_i+1}$$

$$\leq m_i \sum_{j=1}^m \frac{1}{j}$$

因为第 i 轮中有 $m-m_i$ 个对陈旧页面的引用。注意这个求和称为第 m 个调和数, 用 H_m 表示, 可得

$$n_i \leq m_i H_m$$

因此, Marker策略进行的页面替换的期望数至多为

$$U = \sum_{i=1}^k m_i (H_m + 1) = (H_m + 1) \sum_{i=1}^k m_i$$

于是, Marker策略的竞争率至多为

$$\frac{U}{L} = \frac{(H_m + 1) \sum_{i=1}^k m_i}{(1/2) \sum_{i=1}^k m_i} = 2(H_m + 1)$$

利用 H_m 的近似值, 即 $H_m \leq \log m$, Marker策略的竞争率至多为 $2 \log m$ 。

因此, 竞争性分析表明Marker策略是相当有效的。

673

14.3.2 拍卖策略

在这一节里, 我们显示如何把竞争性分析用于某些与Web拍卖有关的简单算法问题。一般设置是我们有一件物品, 像一件古董、艺术品或珠宝, 价值是介于1~ B 美元之间的整数, 我们想要在Web上进行拍卖。对于这种特殊拍卖, 我们必须以在线的方式, 预先给算法 A 提供能够接受的出价, 使得算法 A 在下次显示之前(如果存在下次出价), 能够接受或拒绝当前出价。我们的目标是对于珍贵的物品, 最大化 A 将接受的价值。

1. 最大化问题的竞争性分析

因为在这个问题中, 我们希望最大化利益, 而不是最小化代价。我们稍微需要重新定义 c 可竞争算法的含义。尤其是, 如果

$$\text{cost}(A, P) \geq \text{cost}(\text{OPT}, P)/c + b$$

则称一个最大化算法是 c 可竞争的, 其中 $b \geq 0$ 为常数。如同最小化的情况, 如果 $b = 0$, 那么称这个最大化算法是严格 c 可竞争的。

然而, 如果我们不知道接受多少次出价, 那么一个确定性算法不会有太多的选择, 而是接受第一次出价, 因为这可能是唯一一次出价。但是, 幸亏有竞争性分析, 如果我们把这个策略与一个知道期望出价的对手相比较, 那么我们必定得到的竞争率为 B 。这个竞争率不佳的原因是由于 $P_1 = (1, B)$ 和 $P_2 = (1)$ 都是有效出价序列的事实。因为我们并不知道是期望 P_1 还是 P_2 , 一旦看到它, 必须出价1美元, 而知道序列是 P_1 的对手等待出价 B 美元。但是, 如果我们知道期望出价数 n , 就能做得更好。

定理14.12 如果已知出价数 n , 那么存在拍卖问题的确定性算法, 其竞争率为 $O(\sqrt{B})$ 。

证明 得到这个竞争率的算法相当简单: 接受第一次出价 $\lfloor \sqrt{B} \rfloor$ 。如果没有出现这个出价, 那么接受最后一次出价。为了分析竞争率, 分两种情况。

- 假定没有高于 $\lfloor \sqrt{B} \rfloor$ 的出价。设 m 是给定的最大出价。那么在最坏情况下, 我们可能接受1美元的出价, 而离线算法接受出价 m 。因此, 得到竞争率为 $m \leq \lfloor \sqrt{B} \rfloor$ 。
- 假定有一个高于 $\lfloor \sqrt{B} \rfloor$ 的出价。设 m 是给定的最大出价。那么在最坏情况下, 我们可能接受 $\lfloor \sqrt{B} \rfloor + 1$ 美元的出价, 而离线算法接受出价 m 。因此, 得到竞争率为 $m / (\lfloor \sqrt{B} \rfloor + 1)$ 。这个值当然小于 \sqrt{B} 。 ■

674

这个竞争率肯定不是很好, 但它可能是确定性算法中最好的。

定理14.13 如果已知出价数 n , 那么不存在拍卖问题的竞争率好于 $\Omega(\sqrt{B})$ 的确定性算法。

证明 设 A 是拍卖问题的确定性算法, 并设 $b = \lfloor \sqrt{B} \rfloor$ 。考虑作用在序列 $P = (b, B)$ 上的在线算法 A 的行为。分两种情况。

- 假定 A 接受 b , 那么对手显然能够达到出价 B ; 因此, 这种情况下的竞争率为 $\Omega(\sqrt{B})$ 。
- 假定 A 不接受 P 中的 b 。因为 A 是在线的, 在它拒绝出价 b 的时刻, 它不能区分它是给定的序列 P , 还是序列 $Q = (b, 1)$ 。因此, 可以考虑 A 在 Q 上的行为。因为 A 不知道它将接受的下次出价, 它必须对 b 做出判定, 那么在这种情况下, A 将被迫接受出价1。因此, 在这种情况下, A 的竞争率是 b , 它是 $\Omega(\sqrt{B})$ 。 ■

因此,对于拍卖问题,确定性算法不能得到很好的竞争率。

2. 随机化门限

我们考虑利用随机化算法求解拍卖问题。在这种情况下,通过利用随机化门限 (randomized thresholding) 技术,可以得到竞争率的更多改进,即使算法预先不知道期望的出价数亦可如此。其思想如下。

算法 A 从集合 $\{0, 1, 2, \dots, \log B\}$ 中随机选择一个整数 i , 接受它所接收的第一个大于或等于 2^i 的出价。正如下定理表明的那样,这个随机化门限算法得到的竞争率为 $O(\log B)$ 。

定理 14.14 随机化门限算法得到的竞争率为 $O(\log B)$ 。

证明 设 P 是给定的出价序列, m 是 P 中的最大出价。那么离线算法达到出价 m 。回忆随机化算法的竞争率基于它的期望代价,这在拍卖问题中,应该看作“有益”,因为我们试图最大化接受的出价。这个 A 接受的期望出价值至少为 $m/(1 + \log B)$, 因为 A 接受 m 的概率为 $1/(1 + \log B)$ 。于是, A 的竞争率至多为 $1 + \log B$ 。 ■

675

在下一小节里,我们表明竞争性分析如何应用到数据结构设计中。

14.3.3 竞争查找树

在这一小节里,我们提出一种自适应的基于树的字典结构,它是平衡的和可竞争的。我们的方法基于存储在树中每个结点中的势能 (potential energy) 参数。当进行更新和查询操作时,树中结点的势能会增加或降低。当结点的势能达到一个门限 (threshold) 级时,则重建以那个结点为根的子树。尽管方法简单,但这种模式是竞争性的。

1. 能量平衡的二叉查找树

回忆字典 (dictionary) 存放有序关键字和元素的数对,支持更新和查询操作。实现字典 ADT 的一般方式是利用一棵平衡二叉查找树,通过局部旋转操作保持平衡。这样的旋转操作通常比较快,但是如果树有辅助结构,则旋转常常较慢。我们利用存储在每个结点中的势能参数和部分重建技术,描述一种不需旋转即可达到平衡的简单树型结构。我们提出的方法并没有利用显式平衡规则。而是利用结点中的势能标记,使它成为自适应的、可竞争的并且证明比前面的方法更简单。

能量平衡树 (energy-balanced tree) 是一棵二叉查找树 T , 树中的每个结点 v 存储一个元素 e , 满足 v 的左子树中的所有元素小于或等于 e , v 的右子树中的所有元素大于或等于 e 。 T 中的每个结点 v 维持参数 n_v , 它表示存储在以 v 为根的子树中的元素个数 (包括 v)。更重要的是, T 中每个结点同时还维持势能参数 p_v 。插入和删除如同在标准的 (不平衡) 二叉查找树中进行一样,只有一处小的修改。每当进行更新操作时,遍历从 T 的根到 T 中某个结点 w 的一条路径,对于这条路径上的每个结点 v , 都使 p_v 的值增 1。如果这条路径上不存在结点 v , 满足 $p_v \geq n_v/2$, 那么我们完成操作。否则, 设 v 是 T 中满足 $p_v \geq n_v/2$ 的最大结点。重建以 v 为根的子树, 使它成为一棵完全二叉树, 且把这棵子树中每个结点 (包括 v 自身) 的势能域置为 0。这是进行更新的完整算法。

这个简单的策略直接蕴涵如下结论。

定理 14.15 能量平衡树最坏情况下的高度为 $O(\log n)$, 在这样的树中进行更新的平摊时间为 $O(\log n)$ 。

证明 对于兄弟结点为 w 的任意结点 v , 只要证明 $n_w < n_v/4$ 就足够了。假定不是这样。那么, 由于最后在 v 结点及 w 的父结点 z 的重新平衡, 在 w 的子树中的删除数加上 v 中子树的插入数必定至少为 $3n_v/4$, 即 $p_z \geq 3n_v/4$ 。现在, $n_z = n_w + n_v < 5n_v/4$ 。因此, $p_z \geq 3n_v/4 > (3/5)n_z$ 。但不可能出现这种情况, 因为只要 $p_z > n_z/2$ 就会重建以 v 为根的子树。

676

上述事实直接蕴涵着查找树 T 的高度是 $O(\log n)$ ；因此，对于任意查找，最坏情况下的时间是 $O(\log n)$ 。此外，每次执行重建计算，重建以结点 v 为根的子树， $p_v \geq n_v/2$ ，即 p_v 是 $O(n_v)$ 。因为需要 $O(n_v)$ 时间重建以 v 为根的子树，这个事实蕴涵着我们可以用 p_v 个以前的操作所存放在结点 v 处的“能量”支付重建计算。因为能量平衡树高为 $O(\log n)$ ，且更新操作把能量存放在根结点到叶结点的路径上，这个事实反过来蕴涵着每个更新操作至多被支付 $O(\log n)$ 次以进行重建。因此，在一棵能量平衡查找树中进行更新操作需要的平摊时间也是 $O(\log n)$ 。■

2. 偏能量平衡查找树

可对势能方法进一步进行扩展，把字典改编为适合有偏分布时的访问和更新。在这种情况下，对树 T 进行增大，使每个结点 v 存放一个访问计数（access count） a_v ，它对存储在 v 中元素的访问次数进行计数。每当查找中访问一个结点时，则增加它的访问计数。我们现在还增加从 v 到根的路径上的每个结点上的势能参数。插入算法保持不变，但现在不论什么时候删除一个结点 v ，从 v 到根的路径上的每个结点上的势能都会增加 a_v 。设 A_v 表示 T 中以 v 为根的子树中所有结点的累计访问计数。任何时候执行重建步骤，结点的势能都会升到其访问值的四分之一还多，即 $p_v \geq A_v/4$ 。在这棵改编的二叉查找树中，我们重建子树，使得按照访问计数，结点几乎是平衡的。也就是说，我们试图用子结点的 A_v 值平衡子结点。确切地讲，存在线性时间的重建树算法（见习题C-3.25），对于父结点为 z 的任意结点 v ，能够保证 $A_z \geq 3A_v/2$ 。对于任何非根结点 v ，利用 \hat{A}_v 表示以 v 为根的子树大小加上存储在 v 的父结点 z 中的数据项的权值（ $A_z = A_v + \hat{A}_w$ ，其中 w 表示 v 的兄弟结点）。

引理14.1 对于兄弟结点为 w 的任何结点 v ， $\hat{A}_w \geq A_v/8$ 。

证明 用反证法。假定 $\hat{A}_w < A_v/8$ 。那么，由于最后在 v 结点及 w 的父结点 z 的重新平衡（当 $\hat{A}_w \geq A'_v/2$ 时，其中 \hat{A}_w 和 A'_v 分别表示原 \hat{A}_w 和 A_v 的值），在 w 的子树中删除的总权值加上在 v 的子树中的插入和访问次数再加上终止于 v 的父结点的访问次数必定至少为 $3A_v/8$ ，即 $p_z \geq 3A_v/8$ 。现在， $A_z = \hat{A}_w + A_v < 9A_v/8$ 。因此， $p_z \geq 3A_v/8 > A_z/3$ 。但不可能出现这种情况，因为只要 $p_z > A_z/4$ 就会重建以 z 为根的子树。■

677

上述事实蕴涵着当前访问频率为 a 的元素存储在查找树中深度为 $O(\log A/a)$ 的结点中，其中 A 是树中所有结点的当前总访问频率，如以下定理表明的那样。这个引理直接蕴涵以下定理。

定理14.16 当前访问频率为 a 的元素存储在深度为 $O(\log A/a)$ 的结点中，其中 A 是所有结点的当前总访问频率。

证明 由引理14.1可知，访问以结点 v 为根的子树中的总计数值比访问 v 的子结点的总计数值至少大常数因子。因此，树中任意结点 v 的深度至多为 $O(\log A/a_v)$ 。■

在分析偏能量平衡树之前，首先建立以下技术性引理。这个引理用于分析在整个操作序列 S 的子序列 S_v 上进行操作所需的时间，其中 S 由访问或更新给定结点 v 上元素的所有操作构成。

引理14.2 设 A_i 表示在 S_v 上进行第 i 次操作之后，出现在（ S 的）动态偏能量平衡树中所有结点的总访问计数。那么

$$\sum_{i=1}^m \log A_i / i$$

为

$$O(m \log \hat{A}/m)$$

其中 $m = |S_v|$, \hat{A} 是 S 中引用的所有元素的总访问计数。

证明 为了便于分析, 假设 m 是 2 的幂, 即 $m = 2^k$, 其中 k 为某个数。注意

$$\begin{aligned} \sum_{i=1}^m \log \frac{A_i}{i} &\leq \sum_{i=1}^m \log \frac{\hat{A}}{i} = \sum_{i=1}^m \log \left(\frac{\hat{A}}{m} \right) \left(\frac{m}{i} \right) \\ &= \sum_{i=1}^m \log \frac{\hat{A}}{m} + \sum_{i=1}^m \log \frac{m}{i} = m \log \frac{\hat{A}}{m} + \sum_{i=1}^m \log \frac{m}{i} \end{aligned}$$

因此, 为了确定引理, 只需要限定上面最后一项 (求和项) 的界限。注意

$$\begin{aligned} \sum_{i=1}^m \log \frac{m}{i} &= \sum_{i=1}^{2^k} \log \frac{2^k}{i} \leq \sum_{i=1}^{2^k} \log \frac{2^k}{2^{\lfloor \log i \rfloor}} = \sum_{i=1}^{2^k} k - \lfloor \log i \rfloor \\ &\leq \sum_{j=1}^k j 2^{k-j} = 2^k \sum_{j=1}^k \frac{j}{2^j} \leq 2 \cdot 2^k = 2m \end{aligned}$$

678

需要上述引理把到目前为止看到的序列中的更新和访问的访问计数与整个序列的访问计数相关联。一个神谕 (我们称之为偏树神谕, *biased-tree oracle*) 预先知道序列, 并能够基于已知的访问计数构造一棵静态树, 使得每次在结点 v 处的访问或更新的运行时间为 $O(\log \hat{A}/\hat{a}_v)$, 其中 \hat{a}_v 表示对结点 v 处的元素的总访问计数。

定理 14.17 在能量平衡查找树中的每个结点 v 处进行更新操作, 达到的平摊性能是 $O(\log \hat{A}/\hat{a}_v)$, 与偏树神谕可以达到的性能之差在一个常数因子内。

证明 设 S 是 n 个字典操作的序列, T 是偏树神谕构建的静态树。考虑访问或更新给定结点 v 处的元素的操作形成的 S 的子序列 S_v 。设 A_i 表示在 S_v 上进行第 i 次操作之后, 出现在 (S 的) 动态适应性能量平衡树中的所有结点的总访问次数。注意进行 S_v 中的第 i 个操作的平摊运行时间与 v 在能量平衡树中的未来深度成正比, 它至多为 $O(\log A_i/i)$ 。因此, 执行 S_v 中所有操作所需的平摊时间至多与

$$\sum_{i=1}^m \log A_i / i$$

成正比。而实现偏树神谕所需的总时间则与

$$m \log \hat{A}/\hat{a}_v = m \log \hat{A}/m$$

成正比, 其中 $m = |S_v|$ 。然而, 由引理 14.2 可知,

$$\sum_{i=1}^m \log A_i / i$$

为

$$O(m \log \hat{A}/m)$$

这蕴涵着 S_v 上的能量平衡方法的时间性能至多比偏树神谕方法达到的时间性能多一个常数因子。

679

于是, 类似结论对于 S 中的所有处理序列都成立。

因此, 尽管这些方法很简单, 但偏能量平衡查找树是有效的和有竞争力的。

14.4 习题

基础题

- R-14.1 详细描述 (a, b) 树上的插入和删除算法。
- R-14.2 假定 T 是一棵多路树，每个内部结点至少有5个子结点，至多有8个子结点。要使 T 是一棵有效 (a, b) 树， a 和 b 应取何值？
- R-14.3 要上一个习题中的树 T 是阶为 d 的 B 树， d 应取何值？
- R-14.4 画出一棵阶为7的 B 树，它由把以下关键字插入（按照这个次序）一棵初始为空的树 T 中产生：
(4, 40, 23, 50, 11, 34, 62, 78, 66, 22, 90, 59, 25, 72, 64, 77, 39, 12)
- R-14.5 显示执行序列的四路外存归并排序每一层的递归，其中序列由上一题给出。
- R-14.6 考虑租赁人难题的一般形式，Alice可以分开购买或租用她的滑雪橇和靴子。假定租用滑雪橇需要 a 美元，而购买滑雪橇需要 b 美元。同样，假定租用靴子需要 c 美元，而购买靴子需要 b 美元。为Alice描述一个2-可竞争在线算法，试图使滑雪代价达到最小，约束条件为她不确定未来要继续滑雪多少次。
- R-14.7 考虑初始为空的高速缓存包含4个页面。对于以下页面请求序列：(2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)，LRU算法会导致多少次错失页面？
- R-14.8 考虑初始为空的高速缓存包含4个页面。对于以下页面请求序列：(2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)，FIFO算法会导致多少次错失页面？
- R-14.9 考虑初始为空的高速缓存包含4个页面。对于以下页面请求序列：(2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)，Marker算法会导致多少次错失页面？显示算法中所做的随机选择。
- R-14.10 考虑初始为空的高速缓存包含4个页面。构造一个存储器请求序列，使得Marker算法经过四轮运行。
- R-14.11 显示递归倍增并行算法计算并行序列(1, 4, 20, 12, 7, 15, 32, 10, 9, 18, 11, 45, 22, 50, 5, 16)前缀的过程。

创新题

- C-14.1 显示如何利用无序序列实现外存中的字典，使得最坏情况下插入只需 $O(1)$ 次传输，查找只需 $O(n/B)$ 次传输，其中 n 是元素个数， B 是可放入磁盘块中的表结点数。
- C-14.2 改变定义红黑树的规则，使得每棵红黑树 T 有一棵对应的 $(4, 8)$ 树，反之亦然。
- C-14.3 描述 B 树插入算法的修改版本，使得每当由于结点 v 的分裂引起上溢时，在 v 的所有兄弟结点中重新分配关键字，满足每个兄弟结点大约存放相同数量的关键字（对 v 的父结点进行分裂可能引起层叠）。利用这种模式，每个块总会被充填的最小比例是多少？
- C-14.4 外存字典的另一种可能的实现方法是利用跳跃表，但需要把连续 $O(B)$ 个结点的组收集到跳跃表中任一层的单独块中。特别定义阶为 d (order- d) 的 B 跳跃表 (B -skip list) (它是跳跃表结构的一种表示)，其中每个块中至少包含 $\lceil d/2 \rceil$ 个表结点，至多包含 d 个表结点。在这种情况下，还选择 d 为适合于一个块的跳跃表某层中表结点的最大数量。描述应该如何为 B 跳跃表修改跳跃表的插入和删除算法，使得结构的期望高度为 $O(\log n / \log B)$ 。
- C-14.5 假定在阶为 d 的 B 树 T 中不使用结点查找函数 $f(d) = 1$ ，而是使用 $f(d) = \log d$ 。现在在 T 中进行查找的渐近运行时间变为多少？
- C-14.6 描述实现队列ADT的外存数据结构，使得执行 n 个enqueue和dequeue操作序列所需的磁盘传输总次数为 $O(n/B)$ 。
- C-14.7 描述如何利用 B 树实现划分（合并寻找）ADT（4.2.2节），使得union和find每个操作至多利用 $O(\log n / \log B)$ 次磁盘传输。
- C-14.8 给定具有整型关键字的 n 个元素的序列 S ，使得 S 中的某些元素着“蓝色”，某些元素着“红色”。此外，如果一个红元素 e 与一个蓝元素 f 的关键字相同，则称它们成对 (pair)。描述找出 S 中所有红-蓝对的有效外存算法。你的算法将进行多少次磁盘传输？

- C-14.9 考虑页面缓存问题, 其中高速缓存可以存储 m 个页面, 从 $m+1$ 个可能页面池中, 给定 n 个请求的序列 P 。描述离线算法的最优策略, 并表明它至多总共导致错失 $m + n/m$ 个页面, 从一个空的高速缓存开始。
- C-14.10 考虑基于最少使用 (least frequency used, LFU) 规则的页面缓存策略, 当请求新页面时, 将替换出高速缓存中最少被访问的页面。如果存在相同情况的页面, LFU就会把在高速缓存中存放时间最长的页面替换出来。证明存在 n 个请求的序列 P , 使得对于 m 个页面的高速缓存, LFU导致 $\Omega(n)$ 次错失页面, 而最优算法只会导致 m 次错失页面。
- 681 C-14.11 证明对于 n 个页面请求的序列, LRU是 m 可竞争的, 其中 m 是高速缓存的大小。
- C-14.12 证明对于 n 个页面请求的序列, FIFO是 m 可竞争的, 其中 m 是高速缓存的大小。
- C-14.13 对于大小为 m 的高速缓存以及长为 n 的访问序列, 按照循环方式反复访问 $m+1$ 个块, 随机策略进行块替换的期望次数是多少 (假设 n 比 m 大得多)?
- C-14.14 当高速缓存大小为 m , 并且要访问 $m+1$ 个可能的页面时, 证明Marker算法是 H_m -可竞争的, 其中 H_m 表示第 m 个调和数。
- C-14.15 说明如何用 $O(\log n)$ 的时间, 归并两个由 n 个不同元素组成的有序数组 A 和 B , 算法基于CREW PRAM模型, 并利用 $O(n/\log n)$ 个处理器。
- C-14.16 证明具有 p 个处理器的EREW PRAM E 可以模拟任何具有 p 个处理器的CREW PRAM C , 使得 C 上的每个并行步骤均可在 E 上用 $O(\log p)$ 的时间模拟。
- C-14.17 描述一个 $O(\log n)$ 时间的并行算法, 它利用 n^2 个处理器, 计算两个 $n \times n$ 矩阵的乘积。
- C-14.18 描述一个CRCW PRAM模型上的 $O(1)$ 时间的并行算法, 它利用 n 个处理器, 计算 n 位的AND。
- C-14.19 描述一个CRCW PRAM模型上的 $O(1)$ 时间的并行算法, 它利用 $O(n^2)$ 个处理器, 计算 n 个数中的最大值。
- C-14.20 *描述一个CRCW PRAM模型上的 $O(\log \log n)$ 时间的并行算法, 它利用 $O(n)$ 个处理器, 计算 n 个数中的最大值。
提示: 在首先把数集合划分成大小均为 \sqrt{n} 的 \sqrt{n} 个组时, 应用并行分治法。
- C-14.21 描述一个CREW PRAM模型上的 $O(\log n)$ 时间的并行算法, 利用 $O(n/\log n)$ 个处理器, 对区间 $[1, c]$ 上的 n 个整数的序列 S 进行排序, 其中 $c > 1$ 为常数。
- C-14.22 描述一个CREW PRAM模型上的 $O(\log n)$ 时间的并行算法, 利用 $O(n)$ 个处理器, 计算平面上 n 个点集的凸包 (见第12章), 这些点按照其 x 坐标有序存储在数组中 (可以假设点的 x 坐标不同)。
- C-14.23 设 A 和 B 是两个都存储 n 个整数的有序数组。描述一个CREW PRAM模型上的 $O(\log n)$ 时间的并行算法, 它利用 $O(n/\log n)$ 个处理器, 把 A 和 B 归并成一个有序数组 C 。
提示: 首先把数组 A 中每一个第 $\lceil \log n \rceil$ 个单元分配给一个处理器开始 (对 B 执行同样的处理), 并针对这个数据项在另一数组中进行二分搜索。

程序设计

- P-14.1 利用 (a, b) 树, 编写一个实现2.5节给出的字典中的所有方法的类, 其中 a 和 b 为整型常数。
- 682 P-14.2 实现B树数据结构, 假设块大小为1000, 且关键字为整数。测试进行字典操作序列所需的“磁盘传输”次数。

14.5 本章注记

Knuth[119]很好地讨论了外存排序和查找, Ullman[203]讨论了数据库系统的外存结构。对研究分层的存储器系统体系结构感兴趣的读者可以参考Burger等人[43]或Hennessy和Patterson[93]的著作。Gonnet和Baeza-Yates的手册[81]比较了不同排序算法的性能, 其中有许多是外存算法。Bayer和McCreight[24]发明了B树, Comer[51]对这个数据结构做了很好的综述。Mehlhorn[148]和Samet[176]的著作对B树及其变体还做了很好的讨论。Aggarwal和Vitter[5]研究了排序和相关问题

的I/O复杂度,建立了上界和下界,包括本章中给出的排序下界。Goodrich等人[87]研究了几个计算几何问题的I/O复杂度。对I/O有效算法的进一步研究感兴趣的读者鼓励研究Vitter[206]的综述文献。

有关一个好的并行算法的设计导引,请参阅JáJá[107]、Reif[172]、Leighton[130]及Akl和Lyons[11]的著作。对于最新并行算法设计进展的更多讨论,请参阅Atallah和Chen[16]及Goodrich[84]的著作中的章节。Brent[39]在1976年介绍了我们称为“Brent定理”的设计模式。Kruskal等人[126]研究了并行前缀问题及其应用,Cole和Vishkin[50]详细研究了这个问题及表序问题。Shiloach和Vishkin[186]给出了归并两个有序表的 $O(\log n)$ 运行时间的算法,它基于CREW PRAM模型,利用 $O(n/\log n)$ 个处理器。Valiant[204]的著作表明利用 $O(n)$ 个处理器,可能得到运行时间为 $O(\log \log n)$ 的算法,其中只对比较次数进行计数。Borodin和Hopcroft[34]的著作表明Valiant的算法事实上可在CREW PRAM模型上实现,其运行时间仍然为 $O(\log \log n)$,并利用 $O(n)$ 个处理器。Ajtai、Komlós和Szemerédi[10]的著作表明,基于EREW PRAM模型,利用 $O(n)$ 个处理器可以对 n 个数排序,最优时间为 $O(\log n)$ 。不幸的是,他们的结果在很大程度上只有理论上的兴趣,因为在时间复杂度中涉及的常数非常大。然而Cole[48]给出一个优雅的 $O(\log n)$ 时间的排序算法,它基于EREW PRAM模型,并利用 $O(n)$ 个处理器,其中涉及的常数因子是合理的。找出凸多边形直径的并行算法是对Shamos[185]提出的并行算法的一种改进。

对进一步研究其他在线算法的竞争性分析感兴趣的读者可以参考Borodin和El-Yaniv[33]的著作或者Koutsoupias和Papadimitriou[124]的论文。Borodin和El-Yaniv的著作讨论了Marker高速缓存算法和在线拍卖算法;Motwani和Raghavan[157]的著作在做了类似讨论后,还讨论了模型化上面给出的Marker算法。竞争查找树的讨论基于Goodrich[85]的一篇论文。Overmars[161]介绍了保持数据结构平衡的部分重建概念。习题C-14.11和C-14.12来自Sleator和Tarjan [188]的著作。

附录 A

有用的数学知识

在本附录中，给出几个有用的数学知识。首先介绍一些组合定义和事实。

A.1 对数和指数

对数函数定义为

$$\log_b a = c, \text{ 如果 } a = b^c$$

对于对数和指数，以下恒等式成立：

$$(1) \log_b ac = \log_b a + \log_b c$$

$$(2) \log_b a/c = \log_b a - \log_b c$$

$$(3) \log_b a^c = c \log_b a$$

$$(4) \log_b a = (\log_c a) / \log_c b$$

$$(5) b^{\log_c a} = a^{\log_c b}$$

$$(6) (b^a)^c = b^{ac}$$

$$(7) b^a b^c = b^{a+c}$$

$$(8) b^a / b^c = b^{a-c}$$

此外，还有

定理A.1 如果 $a > 0$, $b > 0$, 且 $c > a + b$, 那么

$$\log a + \log b \leq 2 \log c - 2$$

自然对数 (natural logarithm) 函数 $\ln x = \log_e x$, 其中 $e = 2.71828\cdots$, 它是以下级数的值:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

此外,

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

$$\ln(1+x) = x - \frac{x^2}{2!} + \frac{x^3}{3!} - \frac{x^4}{4!} + \cdots$$

[685] 还有大量与这些函数有关的常用不等式 (它们可由这些函数导出)。

定理A.2 如果 $x > -1$, 那么

$$\frac{x}{1+x} \leq \ln(1+x) \leq x$$

定理A.3 如果 $0 \leq x < 1$, 那么

$$1+x \leq e^x \leq \frac{1}{1-x}$$

定理A.4 对于任何两个正实数 x 和 n , 有

$$\left(1 + \frac{x}{n}\right)^n \leq e^x \leq \left(1 + \frac{x}{n}\right)^{n+x/2}$$

A.2 整型函数和关系

“向下”取整函数和“向上”取整函数分别定义如下:

(1) $\lfloor x \rfloor$ = 小于或等于 x 的最大整数。

(2) $\lceil x \rceil$ = 大于或等于 x 的最小整数。

对于整数 $a \geq 0$, $b > 0$, 模 (modulo) 运算符定义为

$$a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b$$

阶乘 (factorial) 函数定义为

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1)n$$

二项系数为

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

它等于从 n 个数据项的集合中选择 k 个不同数据项的不同组合 (combination) 数 (与顺序无关)。

名字“二项系数”由二项展开 (binomial expansion) 而得:

$$(a+b)^n = \sum_{k=0}^n \binom{n}{k} a^k b^{n-k}$$

还有如下关系式。

定理A.5 如果 $0 \leq k \leq n$, 那么

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \frac{n^k}{k!}$$

定理A.6 (Stirling近似):

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \varepsilon(n)\right)$$

其中 $\varepsilon(n)$ 是 $O(1/n^2)$ 。

斐波那契级数 (Fibonacci progression) 是一个数值级数, 满足 $F_0 = 0$, $F_1 = 1$ 和 $F_n = F_{n-1} + F_{n-2}$, 其中 $n \geq 2$ 。

686

定理A.7 如果 F_n 由斐波那契级数定义, 那么 F_n 是 $\Theta(g^n)$, $g = (1+\sqrt{5})/2$ 是所谓的黄金分割比率 (golden ratio)。

A.3 求和

有许多关于求和的常用事实。

定理A.8 因式分解求和:

$$\sum_{i=1}^n af(i) = a \sum_{i=1}^n f(i)$$

假定 a 不依赖于 i 。

定理A.9 逆序:

$$\sum_{i=1}^n \sum_{j=1}^m f(i,j) = \sum_{j=1}^m \sum_{i=1}^n f(i,j)$$

求和的一种特殊形式是叠缩和 (telescoping sum):

$$\sum_{i=1}^n (f(i) - f(i-1)) = f(n) - f(0)$$

它常出现在数据结构或算法的平摊分析中。

以下是关于求和的其他一些事实, 它们常出现在数据结构和算法的分析中。

定理A.10

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

定理A.11

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

定理A.12 如果 $k \geq 1$ 是整型常数, 那么

$$\sum_{i=1}^n i^k \text{ 是 } \Theta(n^{k+1})$$

另一个常用的求和是几何求和 (geometric sum)

$$\sum_{i=0}^n a^i$$

[687] 其中 a 是任何固定的实数, 且 $0 < a \neq 1$ 。

定理A.13

$$\sum_{i=0}^n a^i = \frac{1-a^{n+1}}{1-a}$$

其中 a 是任何实数, 且 $0 < a \neq 1$ 。

定理A.14

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$$

其中 a 是任何实数, 且 $0 < a < 1$ 。

还有一种两个公共形式的组合, 称为线性指数 (linear exponential) 求和, 它具有如下展开式:

定理A.15 对于 $0 < a \neq 1$, 且 $n \geq 2$,

$$\sum_{i=1}^n ia^i = \frac{a - (n+1)a^{(n+1)} + na^{(n+2)}}{(1-a)^2}$$

第 n 个调和数 H_n 定义为

$$H_n = \sum_{i=1}^n \frac{1}{i}$$

定理A.16 如果 H_n 是第 n 个调和数, 那么 H_n 是 $\ln n + \Theta(1)$ 。

A.4 有用的数学技术

为了确定一个函数是小 o 还是小 ω , 有时希望应用如下规则。

定理A.17 (L'Hôpital规则) 如果有 $\lim_{n \rightarrow \infty} f(n) = +\infty$, 且 $\lim_{n \rightarrow \infty} g(n) = +\infty$, 那么 $\lim_{n \rightarrow \infty} f(n)/g(n) = \lim_{n \rightarrow \infty} f'(n)/g'(n)$, 其中 $f'(n)$ 和 $g'(n)$ 分别表示 $f(n)$ 和 $g(n)$ 的导数。

在导出求和的上界或下界的过程中, 常常如下分裂一个求和式 (split a summation):

$$\sum_{i=1}^n f(i) = \sum_{i=1}^j f(i) + \sum_{i=j+1}^n f(i)$$

另一种有用的技术是用一个积分限定一个求和界限 (bound a sum by an integral)。如果 f 是非增函数, 那么假设定义如下项:

$$\int_{a-1}^b f(x)dx \leq \sum_{i=a}^b f(i) \leq \int_a^{b+1} f(x)dx$$

参 考 书 目

-
- [1] J.D. Achter and R. Tamassia, "Selected topics in algorithms." Manuscript, 1993.
 - [2] G.M. Adel'son-Vel'skii and Y. M. Landis, "An algorithm for the organization of information," *Doklady Akademii Nauk SSSR*, vol. 146, pp. 263-266, 1962. English translation in *Soviet Math. Dokl.*, **3**, 1259-1262.
 - [3] P. K. Agarwal, "Geometric partitioning and its applications," in *Computational Geometry: Papers from the DIMACS Special Year* (J. E. Goodman, R. Pollack, and W. Steiger, eds.), American Mathematical Society, 1991.
 - [4] P. K. Agarwal, "Range searching," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), ch. 31, pp. 575-598, Boca Raton, FL: CRC Press LLC, 1997.
 - [5] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116-1127, 1988.
 - [6] A.V. Aho, "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 255-300, Amsterdam: Elsevier, 1990.
 - [7] A.V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
 - [8] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.
 - [9] R.K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1993.
 - [10] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in clogn parallel steps," *Combinatorica*, vol. 3, pp. 1-19, 1983.
 - [11] S.G. Akl and K. A. Lyons, *Parallel Computational Geometry*. Prentice Hall, 1993.
 - [12] C. Aragon and R. Seidel, "Randomized search trees," in *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 540-545, 1989.
 - [13] K. Arnold and J. Gosling, *The Java Programming Language*. The Java Series, Reading, Mass.: Addison-Wesley, 1996.
 - [14] S. Arya and D. M. Mount, "Approximate nearest neighbor queries in fixed dimensions," in *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pp. 271-280, 1993.
 - [15] S. Arya and D. M. Mount, "Approximate range searching," in *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pp. 172-181, 1995.
 - [16] M.J. Atallah and D. Z. Chen, "Deterministic parallel computational geometry," in *Handbook of Computational Geometry* (J.-R. Sack and J. Urrutia, eds.), pp. 155-200, Amsterdam: Elsevier Science Publishers B.V. North-Holland, 2000.
 - [17] F. Aurenhammer, "Voronoi diagrams: A survey of a fundamental geometric data structure," *ACM Comput. Surv.*, vol. 23, pp. 345-405, Sept. 1991.
 - [18] B. Awerbuch, "Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems," in *19th ACM Symp. on Theory of Computing*, pp. 230-240, 1987.
 - [19] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*. Reading, Mass.: Addison-Wesley, 2nd ed., 1988.
 - [20] E. Bach and J. Shallit, *Algorithmic Number Theory, Volume I.: Efficient Algorithms*. MIT Press, 1996.

- [21] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Reading, Mass.: Addison-Wesley, 1999.
- [22] O. Baruvka, "O jistém problému minimalním," *Praca Moravske Prirodovedecké Společnosti*, vol. 3, pp. 37-58, 1926. (in Czech).
- [23] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance," *Acta Informatica*, vol. 1, no. 4, pp. 290-306, 1972.
- [24] R. Bayer and McCreight, "Organization of large ordered indexes," *Acta Inform.*, vol. 1, pp. 173-189, 1972.
- [25] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87-90, 1958.
- [26] R.E. Bellman, *Dynamic Programming*. Princeton, NJ: Princeton University Press, 1957.
- [27] J. Benaloh and M. de Mare, "One-way accumulators: A decentralized alternative to digital signatures," in *Advances in Cryptology-EUROCRYPT 93*, vol. 765 of *Lecture Notes in Computer Science*, pp. 274-285, 1993.
- [28] J.L. Bentley, "Multidimensional divide-and-conquer," *Commun. ACM*, vol. 23, no. 4, pp. 214-229, 1980.
- [29] J. L. Bentley, "Programming pearls: Thanks, heaps," *Communications of the ACM*, vol. 28, pp. 245-250, 1985.
- [30] J. L. Bentley, D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *SIGACT News*, vol. 12, no. 3, pp. 36-44, 1980.
- [31] J. L. Bentley and T. A. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Trans. Comput.*, vol. C-28, pp. 643-647, Sept. 1979.
- [32] G. Booch, *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummings, 1994.
- [33] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. New York: Cambridge University Press, 1998.
- [34] A. Borodin and J. E. Hopcroft, "Routing, merging, and sorting on parallel models of computation," *J. Comput. Syst. Sci.*, vol. 30, no. 1, pp. 130-145, 1985.
- [35] C.B. Boyer and U. C. Merzbach, *A History of Mathematics*. New York: John Wiley & Sons, Inc., 2nd ed., 1991.
- [36] R.S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762-772, 1977.
- [37] G. Brassard, "Crusade for a better notation," *SIGACTNews*, vol. 17, no. 1, pp. 60-64, 1985.
- [38] G. Brassard and P. Bratley, *Fundamentals of Algorithmics*. Englewood Cliffs, NJ: Prentice Hall, 1996.
- [39] R. P. Brent, "Fast multiple-precision evaluation of elementary functions," *J. ACM*, vol. 23, pp. 242-251, 1976.
- [40] D. Bressoud and S. Wagon, *A Course in Computational Number Theory*. Key College Publishing, 2000.
- [41] E.O. Brigham, *The Fast Fourier Transform*. Englewood Cliffs, NJ: Prentice-Hall, 1974.
- [42] T. Budd, *An Introduction to Object-Oriented Programming*. Reading, Mass.: Addison-Wesley, 1991.
- [43] D. Burger, J. R. Goodman, and G. S. Sohi, "Memory systems," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 18, pp. 447-461, CRC Press, 1997.
- [44] L. Cardelli and P. Wegner, "On understanding types, data abstraction and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471-522, 1985.
- [45] S. Carlsson, "Average case results on heapsort," *BIT*, vol. 27, pp. 2-17, 1987.
- [46] V. Chvátal, "A greedy heuristic for the set-covering problem," *Math. Oper. Res.*, vol. 4 pp. 233-235, 1979.
- [47] K.L. Clarkson, "Linear programming in $O(n^3 d^2)$ time," *Inform. Process. Lett.*, vol. 22, pp. 21-24, 1986.
- [48] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, vol. 17, no. 4, pp. 770-785, 1988.
- [49] R. Cole, "Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm," *SIAM Journal on Computing*, vol. 23, no. 5, pp. 1075-1091, 1994.
- [50] R. Cole and U. Vishkin, "Faster optimal parallel prefix sums and list ranking," *Inform. Comput.*, vol. 81, pp. 334-352, June 1989.
- [51] D. Comer, "The ubiquitous B-tree," *ACM Comput. Surv.*, vol. 11, pp. 121-137, 1979.

- [52] D. Comer, *Computer Networks and Internets*. Englewood Cliffs, NJ: Prentice Hall, 1997.
- [53] S. Cook, "The complexity of theorem proving procedures," in *30th ACM Symp. on Theory of Computing*, pp. 151-158, 1971.
- [54] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297-301, 1965.
- [55] T.H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
- [56] M. Crochemore and T. Lecroq, "Pattern matching and text compression algorithms," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 8, pp. 162-202, CRC Press, 1997.
- [57] S.A. Demurjian, Sr., "Software design," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 108, pp. 2323-2351, CRC Press, 1997.
- [58] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: an annotated bibliography," *Comput. Geom. Theory Appl.*, vol. 4, pp. 235-282, 1994.
- [59] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for Geometric Representations of Graphs*. Englewood Cliffs, NJ: Prentice Hall, 1998.
- [60] E.W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.
- [61] J. R. Driscoll, H. N. Gabow, R. Shrairaman, and R. E. Tarjan, "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation," *Commun. ACM*, vol. 31, pp. 1343-1354, 1988.
- [62] H. Edelsbrunner, "A note on dynamic range searching," *Bull. EATCS*, vol. 15, pp. 34-40, 1981.
- [63] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, vol. 10 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, West Germany: Springer-Verlag, 1987.
- [64] J. Edmonds, "Matroids and the greedy algorithm," *Mathematical Programming*, vol. 1, pp. 126-136, 1971.
- [65] J. Edmonds and R. M. Karp, "Theoretical improvements in the algorithmic efficiency for network flow problems," *Journal of the ACM*, vol. 19, pp. 248-264, 1972.
- [66] D. F. Elliott and K. R. Rao, *Fast Transform Algorithms, Analyses, and Applications*. New York: Academic Press, 1982.
- [67] I. Z. Emiris and V. Y. Pan, "Applications of FFT," in *Algorithms and Theory of Computation Handbook* (M. J. Atallah, ed.), ch. 17, pp. 17-1-17-30, CRC Press, 1999.
- [68] S. Even, *Graph Algorithms*. Potomac, Maryland: Computer Science Press, 1979.
- [69] R.W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.
- [70] R.W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, vol. 7, no. 12, p. 701, 1964.
- [71] L. R. Ford, Jr. and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ: Princeton University Press, 1962.
- [72] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, pp. 596-615, 1987.
- [73] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *SIAM Journal on Computing*, vol. 30, no. 2, pp. 209-221, 1985.
- [74] R.G. Gallager, P.A. Humblet, and P.M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 66-77, 1983.
- [75] T. E. Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. IT-31, no. 4, pp. 469-472, 1985.
- [76] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman, 1979.
- [77] A. M. Gibbons, *Algorithmic Graph Theory*. Cambridge, UK: Cambridge University Press, 1985.
- [78] S. S. Godbole, "On efficient computation of matrix chain products," *IEEE Transactions on Computers*, vol. C-22, no. 9, pp. 864-866, 1973.
- [79] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Reading, Mass.: Addison-Wesley, 1989.
- [80] S. Golomb and L. Baumert, "Backtrack programming," *Journal of the ACM*, vol. 12, pp. 516-524, 1965.
- [81] G.H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures in Pascal and C*. Reading, Mass.: Addison-Wesley, 1991.

- [82] G.H. Gonnet and J. I. Munro, "Heaps on heaps," *SIAM Journal on Computing*, vol. 15, no. 4, pp. 964-971, 1986.
- [83] J. E. Goodman and J. O'Rourke, eds., *Handbook of Discrete and Computational Geometry*. CRC Press LLC, 1997.
- [84] M. T. Goodrich, "Parallel algorithms in geometry," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), ch. 36, pp. 669-682, Boca Raton, FL: CRC Press LLC, 1997.
- [85] M.T. Goodrich, "Competitive tree-structured dictionaries," in *11th ACM-SIAM Symp. on Discrete Algorithms*, pp. 494-495, 2000.
- [86] M.T. Goodrich, M. Handy, B. Hudson, and R. Tamassia, "Accessing the internal organization of data structures in the JDSL library," in *Proc. Workshop on Algorithm Engineering and Experimentation* (M. T. Goodrich and C. C. McGeoch, eds.), vol. 1619 of *Lecture Notes Comput. Sci.*, pp. 124-139, Springer-Verlag, 1999.
- [87] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 714-723, 1993.
- [88] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Inform. Process. Lett.*, vol. 1, pp. 132-133, 1972.
- [89] R.L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, vol. 7, no. 1, pp. 43-57, 1985.
- [90] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*. Reading, Mass.: Addison-Wesley, 1989.
- [91] L.J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, *Lecture Notes Comput. Sci.*, pp. 8-21, Springer-Verlag, 1978.
- [92] Y. Gurevich, "What does $O(n)$ mean?," *SIGACTNews*, vol. 17, no. 4, pp. 61-63, 1986.
- [93] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann, 2nd ed., 1996.
- [94] K. Hinrichs, J. Nievergelt, and P. Schorn, "Plane-sweep solves the closest pair problem elegantly," *Inform. Process. Lett.*, vol. 26, pp. 255-261, 1988.
- [95] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341-343, 1975.
- [96] C.A.R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, pp. 10-15, 1962.
- [97] D. Hochbaum (ed.), *Approximation Algorithms for NP-Hard Problems*. Boston, MA: PWS Publishers, 1996.
- [98] J. E. Hopcroft and R. E. Tarjan, "Efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, no. 6, pp. 372-378, 1973.
- [99] J. E. Hopcroft and J. D. Ullman, "Set merging algorithms," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 294-303, 1979.
- [100] T. C. Hu, *Combinatorial Algorithms*. Reading, Mass.: Addison-Wesley, 1981.
- [101] T. C. Hu and M. T. Shing, "Computations of matrix chain products, part i," *SIAM Journal on Computing*, vol. 11, no. 2, pp. 362-373, 1982.
- [102] T.C. Hu and M. T. Shing, "Computations of matrix chain products, part ii," *SIAM Journal on Computing*, vol. 13, no. 2, pp. 228-251, 1984.
- [103] B. Huang and M. Langston, "Practical in-place merging," *Communications of the ACM*, vol. 31, no. 3, pp. 348-352, 1988.
- [104] D.A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098-1101, 1952.
- [105] C. Huitema, *Routing in the Internet*. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 2000.
- [106] O. H. Ibarra and C. E. Kim, "Fast approximation algorithms for the knapsack and sum of subset problems," *Journal of the ACM*, vol. 9, pp. 463-468, 1975.
- [107] J. JáJá, *An Introduction to Parallel Algorithms*. Reading, Mass.: Addison-Wesley, 1992.

- [108] V. Jarník, "O jistém problemu minimalním," *Praca Moravské Přírodovědecké Společnosti*, vol. 6, pp. 57-63, 1930. (in Czech).
- [109] D. S. Johnson, "Approximation algorithms for combinatorial problems," *J. Comput. Syst. Sci.*, vol. 9, pp. 256-278, 1974.
- [110] R. E. Jones, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.
- [111] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Doklady Akademii Nauk SSSR*, vol. 145, pp. 293-294, 1962. (In Russian).
- [112] D.R. Karger, P. Klein, and R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *Journal of the ACM*, vol. 42, pp. 321-328, 1995.
- [113] R. Karp, "Reducibility among combinatorial problems of computer computations," in *Complexity of Computer Computations* (E. Miller and J. W. Thatcher, eds.), pp. 88-104, New York: Plenum Press, 1972.
- [114] R. M. Karp and V. Ramachandran, "Parallel algorithms for shared memory machines," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), pp. 869-941, Amsterdam: Elsevier/The MIT Press, 1990.
- [115] P. Kirschenhofer and H. Prodinger, "The path length of random skip lists," *Acta Informatica*, vol. 31, pp. 775-792, 1994.
- [116] P. N. Klein and N. E. Young, "Approximation algorithms," in *Algorithms and Theory of Computation Handbook* (M. J. Atallah, ed.), ch. 34, pp. 34-1-34-19, CRC Press, 1999.
- [117] D.E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 1st ed., 1968.
- [118] D.E. Knuth, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 2nd ed., 1973.
- [119] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 1973.
- [120] D.E. Knuth, "Big omicron and big omega and big theta," in *SIGACT News*, vol. 8, pp. 18-24, 1976.
- [121] D.E. Knuth, *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 3rd ed., 1998.
- [122] D.E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 323-350, 1977.
- [123] N. Koblitz, *A Course in Number Theory and Cryptography*. Springer-Verlag, 1987.
- [124] E. Koutsoupias and C. H. Papadimitriou, "On the k -server conjecture," *Journal of the ACM*, vol. 42, no. 5, pp. 971-983, 1995.
- [125] E. Kranakis, *Primality and Cryptography*. John Wiley and Sons, 1986.
- [126] C. P. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix," *IEEE Trans. Comput.*, vol. C-34, pp. 965-968, 1985.
- [127] J. B. Kruskal, Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer. Math. Soc.*, vol. 7, pp. 48-50, 1956.
- [128] D. T. Lee, "Computational geometry," in *The Computer Science and Engineering Handbook* (A. B. Tucker, Jr., ed.), ch. 6, pp. 111-140, CRC Press, 1997.
- [129] D. T. Lee and F. P. Preparata, "Computational geometry: a survey," *IEEE Trans. Comput.*, vol. C-33, pp. 1072-1101, 1984.
- [130] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Mateo, CA: Morgan-Kaufmann, 1992.
- [131] L. A. Levin, "Universal sorting problems," *Problemy Peredachi Informatsii*, vol. 9, no. 3, pp. 265-266, 1973. In Russian.
- [132] A. Levitin, "Do we teach the right algorithm design techniques?," in *30th ACM SIGCSE Symp. on Computer Science Education*, pp. 179-183, 1999.
- [133] H.R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*. Upper Saddle River, New Jersey: Prentice-Hall, 2nd ed., 1998.

- [134] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading, Mass.: Addison-Wesley, 1997.
- [135] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. Cambridge, Mass./New York: The MIT Press/McGraw-Hill, 1986.
- [136] L. Lovász, "On the ratio of optimal integral and fractional covers," *Discrete Math.*, vol. 13, pp. 383-390, 1975.
- [137] N.A. Lynch, *Distributed Algorithms*. San Francisco: Morgan Kaufmann, 1996.
- [138] J. Matoušek, "Geometric range searching," *ACM Comput. Surv.*, vol. 26, pp. 421-461, 1994.
- [139] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of Algorithms*, vol. 23, no. 2, pp. 262-272, 1976.
- [140] E.M. McCreight, "Priority search trees," *SIAM J. Comput.*, vol. 14, no. 2, pp. 257-276, 1985.
- [141] C. J. H. McDiarmid and B. A. Reed, "Building heaps fast," *Journal of Algorithms*, vol. 10, no. 3, pp. 352-365, 1989.
- [142] C. C. McGeoch, "Analyzing algorithms by simulation: Variance reduction techniques and simulation speedups," *ACM Computing Surveys*, vol. 24, no. 2, pp. 195-212, 1992.
- [143] C.C. McGeoch, "Toward an experimental method for algorithm simulation," *INFORMS Journal on Computing*, vol. 8, no. 1, pp. 1-15, 1996.
- [144] C. C. McGeoch, D. Precup, and P. R. Cohen, "How to find the Big-Oh of your data set (and how not to)," in *Advances in Intelligent Data Analysis*, vol. 1280 of *Lecture Notes in Computer Science*, pp. 41-52, Springer-Verlag, 1997.
- [145] N. Megiddo, "Linear-time algorithms for linear programming in R^3 and related problems," *SIAM J. Comput.*, vol. 12, pp. 759-776, 1983.
- [146] N. Megiddo, "Linear programming in linear time when the dimension is fixed," *J. ACM*, vol. 31, pp. 114-127, 1984.
- [147] K. Mehlhorn, "A best possible bound for the weighted path length of binary search trees," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 235-239, 1977.
- [148] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, vol. 1 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.
- [149] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, vol. 2 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.
- [150] K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, vol. 3 of *EATCS Monographs on Theoretical Computer Science*. Heidelberg, Germany: Springer-Verlag, 1984.
- [151] K. Mehlhorn and S. Näher, *LEDA: a Platform for Combinatorial and Geometric Computing*. Cambridge, UK: Cambridge University Press, 1999.
- [152] K. Mehlhorn and A. Tsakalidis, "Data structures," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 301-341, Amsterdam: Elsevier, 1990.
- [153] R. C. Merkle, "Protocols for public key cryptosystems," in *Proc. Symp. on Security and Privacy*, IEEE Computer Society Press, 1980.
- [154] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology-CRYPTO '89* (G. Brassard, ed.), vol. 435 of *Lecture Notes in Computer Science*, pp. 218-238, Springer-Verlag, 1990.
- [155] M. H. Morgan, *Vitruvius: The Ten Books on Architecture*. New York: Dover Publications Inc., 1960.
- [156] D. R. Morrison, "PATRICIA—practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514-534, 1968.
- [157] R. Motwani and P. Raghavan, *Randomized Algorithms*. New York, NY: Cambridge University Press, 1995.
- [158] D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Reading, Mass.: Addison-Wesley, 1996.
- [159] R. Neapolitan and K. Naimipour, *Foundations of Algorithms Using C++ Pseudocode*. Boston: Jones and Bartlett Publishers, 1998.

-
- [160] J. O'Rourke, *Computational Geometry in C*. Cambridge University Press, 1994.
- [161] M.H. Overmars, *The Design of Dynamic Data Structures*, vol. 156 of *Lecture Notes Comput. Sci.* Heidelberg, West Germany: Springer-Verlag, 1983.
- [162] J. Pach, ed., *New Trends in Discrete and Computational Geometry*, vol. 10 of *Algorithms and Combinatorics*. Springer-Verlag, 1993.
- [163] T. Papadakis, J. I. Munro, and P.V. Poblete, "Average search and update costs in skip lists," *BIT*, vol. 32, pp. 316-332, 1992.
- [164] C. H. Papadimitriou and K. Steiglitz, "Some complexity results for the traveling salesman problem," in *Proc. 8th Annu. ACM Sympos. Theory Comput.*, pp. 1-9, 1976.
- [165] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice Hall, 1982.
- [166] D. Peleg, *Distributed Computing: A Locally-Sensitive Approach*. Philadelphia: SIAM, 2000.
- [167] P. V. Poblete, J. I. Munro, and T. Papadakis, "The binomial transform and its application to the analysis of skip lists," in *Proceedings of the European Symposium on Algorithms (ESA)* pp. 554-569, 1995.
- [168] F.P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. New York, NY: Springer-Verlag, 1985.
- [169] R. C. Prim, "Shortest connection networks and some generalizations," *Bell Syst. Tech. J.* vol. 36, pp. 1389-1401, 1957.
- [170] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668-676, 1990.
- [171] M. O. Rabin, "A probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, 1980.
- [172] J. H. Reif, *Synthesis of Parallel Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc., 1993.
- [173] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120-126, 1978.
- [174] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An analysis of several heuristics for the traveling salesman problem," *SIAM J. on Computing*, vol. 6, pp. 563-581, 1977.
- [175] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Reading, MA: Addison-Wesley, 1990.
- [176] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.
- [177] J. E. Savage, *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.
- [178] R. Schaffer and R. Sedgewick, "The analysis of heapsort," *Journal of Algorithms*, vol. 15 no. 1, pp. 76-100, 1993.
- [179] B. Schneier, *Applied cryptography: protocols, algorithms, and sourcecode in C*. New York John Wiley and Sons, Inc., 1994.
- [180] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. New York, NY, USA: John Wiley and Sons, Inc., second ed., 1996.
- [181] A. Schönhage and V. Strassen, "Schnelle multiplikation grosser zahlen," *Computing*, vol. 7 no. 1, pp. 37-44, 1971.
- [182] R. Sedgewick, *Algorithms*. Reading, MA: Addison-Wesley, 1st ed., 1983.
- [183] R. Sedgewick, *Algorithms in C++*. Reading, MA: Addison-Wesley, 1992.
- [184] R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*. Reading, Mass.: Addison-Wesley, 1996.
- [185] M. I. Shamos, "Geometric complexity," in *Proc. 7th Annu. ACM Sympos. Theory Comput.*, pp. 224-233, 1975.
- [186] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," *J. Algorithms*, vol. 2, pp. 88-102, 1981.
- [187] M. Sipser, *Introduction to the Theory of Computation*. PWS Publishing Co., 1997.
- [188] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, pp. 202-208, 1985.

- [189] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652-686, 1985.
- [190] R. Solovay and V. Strassen, "A fast Monte-Carlo test for primality," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 84-85, 1977.
- [191] R. Solovay and V. Strassen, "Erratum: A fast Monte-Carlo test for primality," *SIAM Journal on Computing*, vol. 7, no. 1, 1978.
- [192] W. Stallings, *Cryptography and network security: principles and practice*. Upper Saddle River, NJ 07458, USA: Prentice-Hall, Inc., second ed., 1999.
- [193] G.A. Stephen, *String Searching Algorithms*. World Scientific Press, 1994.
- [194] R. Tamassia, "Graph drawing," in *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), ch. 44, pp. 815-832, Boca Raton, FL: CRC Press LLC, 1997.
- [195] R. Tamassia, M. T. Goodrich, L. Vismara, M. Handy, G. Shubina, R. Cohen, B. Hudson, R. S. Baker, N. Gelfand, and U. Brandes, "JDSL: The data structures library in Java," *Dr. Dobbs's Journal*, vol. 323, April 2001.
- [196] A. S. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall, 1996.
- [197] R. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. Comput.*, vol. 14, pp. 862-874, 1985.
- [198] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146-160, 1972.
- [199] R.E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. Comput. System Sci.*, vol. 18, pp. 110-127, 1979.
- [200] R. E. Tarjan, *Data Structures and Network Algorithms*, vol. 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.
- [201] R. E. Tarjan, "Amortized computational complexity," *SIAM J. Algebraic Discrete Methods*, vol. 6, no. 2, pp. 306-318, 1985.
- [202] G. Tel, *Introduction to Distributed Algorithms*. New York: Cambridge University Press, 1994.
- [203] J. D. Ullman, *Principles of Database Systems*. Potomac, MD: Computer Science Press, 1983.
- [204] L. Valiant, "Parallelism in comparison problems," *SIAM J. Comput.*, vol. 4, no. 3, pp. 348-355, 1975.
- [205] J. van Leeuwen, "Graph algorithms," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A. Algorithms and Complexity, pp. 525-632, Amsterdam: Elsevier, 1990.
- [206] J. S. Vitter, "Efficient memory access in large-scale computation," in *Proc. 8th Sympos. Theoret. Aspects Comput. Sci.*, Lecture Notes Comput. Sci., Springer-Verlag, 1991.
- [207] J. S. Vitter and P. Flajolet, "Average-case analysis of algorithms and data structures," in *Algorithms and Complexity* (J. van Leeuwen, ed.), vol. A of *Handbook of Theoretical Computer Science*, pp. 431-524, Amsterdam: Elsevier, 1990.
- [208] J. Vuillemin, "A unifying look at data structures," *Commun. ACM*, vol. 23, pp. 229-239, 1980.
- [209] S. Warshall, "A theorem on boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11-12, 1962.
- [210] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347-348, 1964.
- [211] D. Wood, *Data Structures, Algorithms, and Performance*. Reading, Mass.: Addison-Wesley, 1993.
- [212] F. F. Yao, "Computational geometry," in *Algorithms in Complexity* (R. A. Earnshaw and B. Wyvill, eds.), pp. 345-490, Amsterdam: Elsevier, 1990.
- [213] C.K. Yap, *Fundamental Problems in Algorithmic Algebra*. Oxford University Press, 1999.

索引

索引中的页码为英文原书的页码, 与书中边栏的页码一致。

δ -approximation (δ -近似), 618
 c -incremental (c -增量), 593
2SAT (2元合取范式可满足性), 607, 640
3SAT (3元合取范式可满足性), 607, 608

A

abstract data type (抽象数据类型)
 dictionary (字典), 114-115, 141
 graph (图), 289-295
 list (表), 69-70
 partition (划分), 227-234
 priority queue (优先队列), 94-95, 112-113
 queue (队列), 61
 sequence (序列), 73-74
 set (集合), 225-234
 stack (栈), 57
 string (串), 419-420
 tree (树), 77-78
 vector (向量), 65
(a, b) tree ((a, b) 树), 650-652
 depth property (深度性质), 650
 size property (大小性质), 650
accepting a string (接受一个串), 594
access control list (访问控制表), 472
accounting method (会计方法), 36-37
Achter, 510
Ackermann function (Ackermann函数), 234, 256
acyclic (无环的), 316
additive inverse (加可逆), 458
Adel'son-Vel'skii, 216
adjacency list (邻接表), 296, 299
adjacency matrix (邻接矩阵), 296, 301
adjacent (邻接的), 290
Adleman, 476
Agarwal, 590
Aggarwal, 683
Aho, 137, 216, 256, 450, 510, 642
Ahuja, 338, 379, 414

Ajtai, 683
Akl, 683
algorithm (算法), 4
algorithm analysis (算法分析), 8-33
 average case (平均情况), 11
 worse case (最坏情况), 11
alphabet (字母表), 420
amortization (平摊方法), 34-41, 80, 133, 191-194, 215,
 227-234, 427, 625-626, 676-679
 accounting method (会计方法), 36-37
 potential function (势能函数), 37-38
ancestor (祖先), 75, 315
anchor point (锚点), 578-581
antipodal (对映的), 665
antisymmetric property (反对称性), 94
anycast (任意播), 544
approximation algorithm (近似算法), 618
approximation algorithm (近似算法), 618-626
Aragon, 590
arc (弧), 289
Archimedes, 4, 54
Ariadne, 288
Arnold, 137
array splitting (数组分裂), 663
art gallery guarding (艺术画廊保护), 283
Arya, 590
asymmetric relation (非对称关系), 289
asymptotic notation (渐近符号), 13-33
 big-Oh (大 O), 13-16
 big-Omega (大 Ω), 16
 big-Theta (大 Θ), 16
 little-oh (小 o), 18
 little-omega (小 ω), 18
Atallah, 683
auction algorithm (拍卖算法), 674-675
audit trail (审计线索), 115
augmenting cycle (增大回路), 398
augmenting path (增大路径), 388

Aurenhammer, 590
 authenticated dictionary (认证字典), 482
 AVL tree (平衡二叉树), 152-158, 206, 569
 balance factor (平衡因子), 206
 height-balance property (高度平衡性质), 152
 Awerbuch, 544

B

Baase, 510
 back edge (后边), 305, 318, 320, 336
 backtracking (回溯), 303, 627-631, 642
 Baeza-Yates, 216, 256, 450, 683
 balance factor (平衡因子), 206
 balanced search tree (平衡查找树), 162
 Barůvka, 379
 Barůvka's algorithm (Barůvka算法), 369-372, 528-529
 Baumert, 642
 Bayer, 216, 683
 Bellman, 284, 379
 Bellman-Ford algorithm (Bellman-Ford算法), 349-351
 Bentley, 137, 284, 590
 Bertrand's Postulate (Bertrand假设), 126
 best-first search (最佳首次查找), 632
 BFS, *see* breadth-first search (BFS, 见广度优先搜索)
 BFS tree (BFS树), 315
 biconnected (双连通), 307
 biconnected component (双连通分量), 307
 big integer (大整数), 270
 big-Oh notation (大O符号), 13-16, 54
 big-Omega notation (大 Ω 符号), 16
 big-Theta notation (大 Θ 符号), 16
 Binary Euclid's Algorithm (二进制欧几里得算法), 457
 binary search (二分搜索), 142-145
 binary search tree (二叉查找树), 145-151
 insertion (插入), 148
 removal (删除), 149-150
 rotation (旋转), 155
 trinode restructure (三结点重构), 155
 binary tree (二叉树), 76, 84-92, 220
 complete (完全的), 99
 left child (左子结点), 76
 level (层), 84
 linked structure (链表结构), 92
 proper (性质), 76
 right child (右子结点), 76
 vector representation (向量表示), 90-91
 binomial expansion (二项展开), 686
 bipartite graph (二分图), 396
 bit commitment (比特承诺), 483
 bit vector (比特向量), 252

blocking (分块), 647
 Booch, 137
 Boolean circuit (布尔电路), 597
 bootstrapping (自展法), 161
 Borodin, 683
 bottleneck (瓶颈), 395
 boundary node (边界点), 551
 Boyer, 54, 450
 branch-and-bound (分枝限界法), 632-634, 639, 641, 642
 Brassard, 54, 642
 Bratley, 642
 breadth-first search (广度优先搜索), 313-316, 320, 523-527
 Brent, 683
 Brent's theorem (Brent定理), 659, 660
 Brigham, 510
 broadcast routing (广播路由), 530-531
 brute force (蛮力法), 420
 brute-force pattern matching (蛮力模式匹配), 420
 B-tree (B树), 652-653
 bubble-sort (冒泡排序), 137
 bucket array, (桶数组), 116
 bucket-sort (桶排序), 241-242
 Budd, 137
 Burger, 683

C

cache (高速缓存), 645
 cache line (高速缓存线), 647
 caching algorithm (高速缓存算法), 668-673
 call-by-value (按值调用), 60
 capacity rule (容量定律), 383
 Cardelli, 137
 Carlsson, 137
 Carmichael number (Carmichael数), 466
 Cartesian coordinate (Cartesian坐标), 572
 Cartesian tree (Cartesian树), 590
 Catalan number (Catalan数), 275
 certificate (证书), 486, 596
 certificate authority (证书授权), 486
 certificate revocation list (证书撤销表), 486
 character-jump heuristic (字符跳跃的启发式搜索), 422
 Chen, 683
 Chernoff bound (Chernoff界限), 253
 child (子结点), 75
 children (子结点 (复数)), 75
 Chomsky normal form (Chomsky范式), 284
 Chvátal, 642
 CIRCUIT-SAT (电路可满足性), 598, 600-602, 605
 Clarkson, 256
 clauses (子句), 605

- clearable table (可清空表), 34
- CLIQUE (团), 610, 638
- clique (团), 610
- clockwise (顺时针), 574
- closed form (封闭形式), 12
- closest pair (最近点对), 568-571
- closure (闭包), 357
- CNF *see* conjunctive normal form (CNF, 见合取范式),
- CNF-SAT (合取范式可满足性), 605-607, 628, 629, 640, 642
- co-*NP*, 595, 597
- coefficient form (系数形式), 488
- Cole, 450, 683
- collinear (共线), 574
- collision resolution (冲突解决方法), 116, 120-127
- collision-resistant (抗冲突), 481
- Comer, 544, 683
- comparator (比较器), 95, 218, 579, 583
- radial (射线的), 579, 580, 583
- competitive search tree (竞争查找树), 676-679
- complement (补), 595
- complete binary tree (完全二叉树), 99
- complexity class (复杂类), 595
- component design (分量设计), 603
- composite (合数), 453
- compositeness witness function (合性证据函数), 466
- compression map (压缩映射), 117, 119
- computational geometry (计算几何), 548-590
- closest pair (最近点对), 568-571
- convex hull (凸包), 572-583
- degeneracy (退化), 575, 583
- orientation (方位), 574-575
- plane sweep (平面扫描), 566-571
- proximity (接近程度), 568
- representation (表示), 572-573
- segment intersection (线段相交), 565-568
- conditional probability (条件概率), 29
- confidentiality (秘密), 452
- congruence (同余), 454
- congruent (同余的), 454
- conjunctive normal form (合取范式), 605, 607
- connected component (连通分量), 292, 306, 315
- conservation rule (守恒定律), 383
- container (容器), 137
- contradiction (反证法), 25
- contrapositive (逆否命题法), 25
- convex hull (凸包), 572-583, 590
- gift wrapping (礼品包扎), 578-579
- Graham scan algorithm (Graham扫描算法), 580-583
- convexity (凸性), 573, 576
- convolution (卷积), 488
- Convolution Theorem (卷积定理), 492
- Cook, 642
- Cook-Levin Theorem (Cook-Levin定理), 600
- Cooley, 510
- coordinate (坐标), 549
- core memory (核心存储器), 645
- Cormen, 216, 338, 379, 414, 510, 642
- cost (代价), 398
- counterclockwise (逆时针), 574
- Crochemore, 450
- cross edge (交叉边), 315, 318, 320
- cryptography (密码学), 471-480
- El Gamal cryptosystem (El Gamal加密系统), 479-480
- RSA cryptosystem (RSA加密系统), 476-478
- cursor (光标), 70
- cut (割), 385
- cyber-dollar (计算机元), 36
- cycle (回路, 环), 292
- directed (有向的), 292
- cryptograph (加密)
- public-key (公钥), 475-480
- ## D
- DAG, *see* directed acyclic graph (DAG, 见有向无环图)
- data integrity (数据完整性), 452, 472
- data structure (数据结构), 4
- secondary (辅助), 161
- decision problem (判定问题), 594
- decision tree (判定树), 239
- decorator pattern (修饰模式), 329-332
- decrease-and-conquer, *see* prune-and-search (减小-求解法, 见剪枝-查找法)
- decryption (解码, 译码), 471
- degree (度), 290, 488
- Delaunay triangulation (Delaunay三角剖分), 590
- DeMorgan's Law (DeMorgan定律), 25
- Demurjian, 137
- depth (深度), 79-80
- depth-bounded tree (深度有界树), 159
- depth-first search (深度优先搜索), 303-306
- descendent (后代), 75, 315
- design pattern (设计模式)
- amortization (平摊方法), 39-41
- brute force (蛮力法), 420
- comparator (比较器), 95
- decorator (修饰), 329-332
- divide-and-conquer (分治法), 219-221, 235, 263-273
- dynamic programming (动态规划), 274-281, 444-446
- greedy method (贪心法), 442

iterator (迭代器), 74
 locator (定位器), 112-113
 position (位置), 68
 prune-and-search (剪枝-查找), 245-247
 template method (模板方法), 332
 destination (目的点), 290
 DFS, *see* depth-first-search (DFS, 见深度优先搜索)
 DFS tree (DFS树), 305
 DFT, *see* Discrete Fourier Transform (DFT, 见离散傅里叶变换)
 Di Battista, 137, 338
 diameter (直径), 135, 523, 665
 dictionary (字典), 114-127, 141-216
 (2,4) tree ((2,4)树), 163-169
 abstract data type (抽象数据类型), 114-115, 141
 AVL tree (AVL树), 152-158
 binary search tree (二叉查找树), 145-151
 hash table (散列表), 116-127
 log file (日志文件), 115
 lookup table (查找表), 142-145
 ordered (有序的), 114, 141
 red-black tree (红黑树), 170-184
 skip list (跳跃表), 195-202
 unordered (无序的), 114, 115
 update operation (更新操作), 148, 149, 154, 157, 198, 199
 Diffie-Hellman key exchange (Diffie-Hellman密钥交换), 487
 digital signature (数字签名), 471
 El Gamal, 480
 RSA, 477
 digraph (有向图), 316
 Dijkstra, 379
 Dijkstra's algorithm (Dijkstra算法), 342-348, 373-376
 directed acyclic graph (有向无环图), 325-327
 directed cycle (有向回路), 316
 directed DFS tree (有向DFS树), 318
 discovery edge (发现边), 305, 315, 318, 320
 Discrete Fourier Transform (离散傅里叶变换), 491-494
 discrete logarithm (离散对数), 479
 disjunctive normal form (析取范式), 606
 distance (距离), 341
 distance vector (距离矢量), 532
 distance vector algorithm (距离矢量算法), 532-533
 distributed algorithm (分布式算法), 514-517
 message-passing model (消息传递模型), 514-516
 distributed computing (分布式计算)
 asynchronous model (异步模型), 515
 synchronous model (同步模型), 515
 divide-and-conquer (分治法), 219-221, 235, 263-273, 659

division method (除法散列法), 119
 DNF, *see* disjunctive normal form (DNF, 见析取范式)
 DNF-DISSAT, 638
 DNF-SAT, 638
d-node (*d*-结点), 159
 double black (双黑色), 177
 double hashing (双散列法), 124
 double red (双红色), 172
 down-heap bubbling (堆中向下冒泡), 104, 109
 dynamic programming (动态规划), 274-281, 320, 443-446

E

Edelsbrunner, 590
 edge (边), 289, 573
 destination (目的点), 290
 end vertice (端点), 290
 incident (依附), 290
 multiple (多), 291
 origin (源点), 290
 outgoing (出), 290
 parallel (并行), 291
 self-loop (自循环), 291
 edge capacity (边容量), 383
 edge list (边表), 296, 298
 edit distance (编辑距离), 449
 Edmonds, 284, 414
 Edmonds-Karp algorithm (Edmonds-Karp算法), 393-395
 Edmonds-Karp augmentation (Edmonds-Karp增大路径), 393
 El-Yaniv, 683
 Elliott, 510
 El Gamal cryptosystem (El Gamal密码系统), 479-480
 Emiris, 510
 encryption scheme (加密模式), 473
 end vertice (端点), 290
 endpoint (端点), 290
 energy-balanced tree (能量平衡树), 676-679
 equivalence class (等价类), 308
 equivalence relation (等价关系), 308
 Euclid's algorithm (欧几里得算法), 455-457
 binary (二进制), 457
 extended (扩展的), 464-465
 Euler pseudo-prime (欧拉伪素数), 469
 Euler tour (欧拉路径), 335, 338
 Euler tour traversal (欧拉路径遍历), 87, 137
 Euler's Theorem (欧拉定理), 461
 Even, 338, 379, 414
 event (事件), 28
 expected value (期望值), 29
 exponent (指数), 23

Extended Euclid's Algorithm (扩展欧几里得算法), 464
 external memory (外存), 644-656, 683
 external memory algorithm (外存算法), 645-656
 external memory sorting (外存排序), 654-656

F

factorial (阶乘), 686
 failure function (失效函数), 425
 fairness (公平性), 515
 Fast Fourier Transform (快速傅里叶变换), 488-507, 510
 Fermat's Little Theorem (Fermat小定理), 459
 FFT, *see* Fast Fourier Transform (FFT, 见快速傅里叶变换)
 Fibonacci progression (斐波那契级数), 687
 FIFO (先进先出), 61, 656
 first-in first-out (先进先出), 61
 Flajolet, 54
 flip a random coin (投掷一枚随机硬币), 483
 flow, *see* network flow (流, 见网络流)
 flow network (流网络), 383
 Floyd, 137
 Floyd-Warshall algorithm (Floyd-Warshall算法), 320, 338
 Ford, 379
 Ford-Fulkerson algorithm (Ford-Fulkerson算法), 387-395
 forest (森林), 292
 forward edge (前向边), 318
 frame (框架), 59
 Fulkerson, 414
 fully polynomial-time approximation scheme (完全多项式时间的近似模式), 619
 Fundamental Theorem of Arithmetic (基本运算定理), 453
 fusion (融合), 167, 651, 653

G

Gabow, 256
 Gallego, 544
 garbage collection (垃圾收集), 323-324
 mark-sweep (标记-扫描), 323
 Garey, 642
 Gauss, 22
 Gavril, 642
 GCD, *see* greatest common divisor (GCD, 见最大公因子)
 generator (生成元), 462
 generic merge algorithm (泛型归并算法), 226
 geometric sum (几何求和), 687
 Gibbons, 338, 379, 414
 gift wrapping (礼品包扎), 578-579
 Godbole, 284
 Golberg, 137
 golden ratio (黄金分割比率), 687

Golomb, 642
 Gonnet, 137, 216, 256, 683
 Goodman, 590
 Goodrich, 590, 683
 googol (10的100次方), 17
 Gosling, 137
 Graham, 54, 379, 590
 Graham scan algorithm (Graham扫描算法), 580-583
 grammar (语法), 284
 graph (图), 288-338, 340-379
 abstract data type (抽象数据类型), 289-295
 acyclic (无环的), 316
 bipartite (二分), 396
 breadth-first search (广度优先查找), 313-316, 318-320
 connected (连通的), 292, 315
 data structure (数据结构), 296-301
 adjacency list (邻接表), 299-301
 adjacency matrix (邻接矩阵), 301
 edge list (边表), 296-298
 dense (稠密的), 321
 depth-first search (深度优先查找), 303-306, 318-320
 digraph (有向图), 316
 directed (有向的), 289, 290, 316-327
 acyclic (无环的), 325-327
 strongly connected (强连通的), 316
 method (方法), 293-295
 mixed (混合的), 290
 reachability (可达性), 316-317, 320-321
 shortest path (最短路径), 320-321
 simple (简单的), 291
 traversal (遍历), 303-316
 undirected (无向的), 289, 290
 weighted (带权的), 340-379
 greatest common divisor (最大公因子), 454-457
 Euclid's algorithm (Euclid算法), 455-457
 greedy method (贪心法), 259-262, 341, 342, 442
 greedy-choice (贪心选择), 259, 442
 group (群), 461
 guess-and-test (猜测-测试), 266-267
 Guibas, 216
 Guttag, 137

H

Haken, 284
 HAMILTONIAN-CYCLE (哈密尔顿回路), 597, 615, 617, 641, 642
 Harmonic number (调和数), 626
 harmonic number (调和数), 688
 hash code (散列编码), 117, 118
 hash function (散列函数), 117, 124-126

2-universal (2-通用的), 125
 one-way (单向), 481
 hash table (散列表), 116-127
 bucket array (桶数组), 116
 capacity (容量), 116
 chaining (链地址法), 121
 clustering (聚集), 124
 collision (冲突), 116
 collision resolution (冲突解决方法), 120-127
 double hashing (双散列), 124
 linear probing (线性探测), 123
 open addressing (开放定址法), 124
 quadratic probing (二次探测), 124
 secondary clustering (二次聚集), 124
 universal hashing (通用散列), 125-127
 hash value (散列值), 118
 header (头结点), 70
 heap (堆), 99-111
 bottom-up construction (自底向上构造), 109-111
 heap-order property (堆序性质), 99
 heap-sort (堆排序), 107-111, 218
 height (高度), 79-80
 height-balance property (高度平衡性质), 152, 154, 157
 Hell, 379
 Hennessy, 683
 hierarchical (分层的), 56
 Hinrichs, 590
 Hirschberg, 284
 Hoare, 256
 Hochbaum, 642
 Hopcroft, 137, 216, 256, 338, 510, 642, 683
 Horner's method (Horner方法), 52
 Horner's rule (Horner规则), 488
 Hu, 284
 Huang, 256
 Huffman, 450
 Huffman coding (赫夫曼编码), 440-441
 Huitema, 544

I

Ibarra, 642
 identity matrix (单位矩阵), 492
 in-degree (入度), 290
 in-place (原位), 248, 324
 incidence container (关联容器), 299
 incidence matrix (关联矩阵), 612
 incident (关联的), 290
 incoming edges (入边), 290
 independent (独立的), 28, 30
 independent set (独立集), 284

INDEPENDENT-SET, 640
 index, *see* discrete logarithm (下标, 见离散对数)
 induction (归纳法), 25-26
 inorder traversal (中序遍历), 146, 150, 155
 input size (输入大小), 593
 insertion-sort (插入排序), 98, 218
 inside node (内结点), 551
 integer multiplication (整数相乘), 270-272
 internal memory (内存), 645
 inverse shuffle (逆混洗), 502
 inversion (逆), 254
 IP router (IP路由器), 514
 item (数据项), 114
 iterative substitution (迭代替换), 264
 iterator (迭代器), 74

J

Jacobi symbol (Jacobi符号), 468
 JáJá, 137, 683
 Jarník, 379
 Java Virtual Machine (Java虚拟机), 64, 137
 JDLS, 137
 Johnson, 642
 Jones, 338
 Josephus problem (约瑟夫环问题), 133

K

Karatsuba, 284
 Karger, 379
 Karp, 137, 414, 642
 k -D tree (k -D树), 563-564
 key (密钥), 94, 114, 115, 159
 key transfer (密钥传输), 475
 Kim, 642
 Klein, 379, 642
 KNAPSACK (背包), 614, 619-621, 634, 639-642
 knapsack problem (背包问题), 259-260, 278-281
 Knuth, 54, 137, 216, 256, 338, 450, 683
 Komlós, 683
 Kosaraju, 338
 Koutsoupias, 683
 Kruskal, 379, 683
 Kruskal's algorithm (Kruskal算法), 362-366

L

L'Hôpital's Rule (L'Hôpital规则), 688
 Landis, 216
 Langston, 256
 language (语言), 594
 last node (最后一个结点), 99

last-in first-out (后进先出), 57
 LCS, *see* longest common subsequence (LCS, 见最长公共子序列)
 leader election (领导人选举), 517-523
 leaves (叶结点), 75
 Lecroq, 450
 LEDA, 137
 Lee, 590
 left child (左子结点), 76
 left subtree (左子树), 76
 left turn (左旋转), 574
 Legendre symbol (Legendre符号), 467
 Leighton, 683
 Leiserson, 216, 338, 379, 414, 510, 642
 level (层), 84, 313
 level numbering (层编号), 90
 level order traversal (层序遍历), 134
 Levin, 642
 Lewis, 642
 lexicographical (字典的), 242
 LIFO (后进先出), 57
 Lindholm, 137
 line (直线), 572
 linear exponential (线性指数的), 688
 linear probing (线性探测), 123
 linearity of expectation (线性期望), 29, 246
 link component (连接分量), 310
 link relation (连接关系), 308
 link-state algorithm (链路状态算法), 534-535
 linked list (链表)
 doubly linked (双向链表), 70-73
 linked structure (链表结构), 92
 Liskov, 137
 list (表), 68-72, 115
 abstract data type (抽象数据类型), 69-72
 list ranking (表序), 661
 literal (文字), 605
 little-oh notation (小 o 符号), 18
 little-omega notation (小 ω 符号), 18
 live object (活动对象), 323
 load factor (装填因子), 122
 local replacement (局部替换法), 603
 locality-of-reference (引用局部性), 647
 locator pattern (定位器模式), 112-113
 log file (日志文件), 115, 121
 logarithm (对数), 23, 685
 natural (自然的), 685
 longest common subsequence (最长公共子序列), 443-446
 looking-glass heuristic (照镜子启发式搜索), 422
 lookup table (查找表), 142-145
 loop invariant (循环不变式), 27

Lovása, 642
 LRU (最近最久未使用算法), 656
 Lynch, 544
 Lyons, 683

M

machine scheduling (单机调度), 283
 Magnanti, 338, 379, 414
 main memory (主存), 645
 mark-sweep algorithm (标记-扫描算法), 323
 Marker strategy (Marker策略), 671
 master method (主方法), 268-270
 matching (匹配), 396
 matrix chain-product (矩阵链乘), 274-277
 matrix closure (矩阵闭包), 357-359
 matroid theory (拟阵理论), 284
 Max-Flow, Min-Cut Theorem (最大流、最小割定理), 389
 maximal independent set (最大独立集), 338
 maximum bipartite matching (最大二分匹配), 396-397
 maximum flow, *see* network flow (最大流, 见网络流)
 McCreight, 450, 590, 683
 McDiarmid, 137
 McGeoch, 54
 median (中值), 245
 Megiddo, 256
 Mehlhorn, 216, 338, 379, 414, 590, 683
 memory hierarchy (存储器层次结构), 645
 memory management (存储器管理), 323, 646-648, 668-673
 merge-sort (归并排序), 219-224
 multi-way (多路), 654-656
 tree (树), 220
 mergeable heap (可归并堆), 215
 Merzbach, 54
 message-passing model (消息传递模型), 514-516
 method stack (方法栈), 59-60
 minimum cut (最小割), 386
 minimum spanning tree (最小生成树), 360-372, 528
 Barůvka's algorithm (Barůvka算法), 369-372, 528-529
 Kruskal's algorithm (Kruskal算法), 362-366, 372
 Prim-Jarnik algorithm (Prim-Jarnik算法), 366-367, 372
 minimum-cost flow (最小代价流), 398-405
 Minotaur, 288
 modular arithmetic (模运算), 62, 126, 454, 458-462, 686
 modular exponentiation (模指数运算), 462-464
 modular multiplicative inverse (模乘法逆元), 464
 modulus (模数), 454
 Moore, 450
 Morris, 450

Morrison, 450
 Motwani, 216, 256, 683
 Mount, 590
 MST, *see* minimum spanning tree (MST, 见最小生成树)
 multi-way search tree (多路查找数), 159
 multi-way tree (多路树), 159-162
 multicast routing (多播路由), 535-540
 multiplicative group (乘法群), 461
 multiplicative inverse (乘法逆元), 458, 464
 multiprogramming (多道程序设计), 63
 Munro, 137
 mutually independent (相互独立), 28

N

Naimipour, 642
 natural logarithm (自然对数), 685
 Neapolitan, 642
 network flow (网络流), 382-414
 augmenting cycle (增大回路), 398
 augmenting path (增大路径), 388
 backward edge (后向边), 385
 bottleneck (瓶颈), 395
 capacity rule (容量定律), 383
 conservation rule (守恒定律), 383
 cut (割), 385-386
 cut capacity (割容量), 386
 edge capacity (边容量), 383
 Edmonds-Karp algorithm (Edmonds-Karp算法), 393-395
 flow across a cut (穿越割的流), 386
 flow network (流网络), 383-385
 flow value (流的值), 384
 Ford-Fulkerson algorithm (Ford-Fulkerson算法), 387-395
 forward edge (前向边), 385
 Max-Flow, Min-Cut Theorem (最大流、最小割定理), 389
 maximum flow problem (最大流问题), 384, 387
 minimum cut (最小割), 386
 minimum-cost flow (最小代价流), 398-405
 residual capacity (剩余容量), 387
 residual distance (剩余距离), 393
 residual graph (剩余图), 392
 network protocol stack (网络协议栈), 514
 networking protocol stack (网络协议栈), 513-514
 application layer (应用层), 513
 data-link layer (数据链路层), 513
 network layer (网络层), 513
 physical layer (物理层), 513
 transport layer (传输层), 513
 node (结点), 68, 75, 77, 289
 ancestor (祖先), 75

balanced (平衡的), 154
 boundary (边界), 551
 child (子结点), 75
 descendent (后代), 75
 external (外部的), 75
 inside (内), 551
 internal (内部的), 75
 outside (外), 551
 parent (父结点), 75
 redundant (冗余的), 433
 root (根), 75
 sibling (兄弟结点), 75
 size (规模, 大小), 191
 unbalanced (不平衡的), 154
 nontree edge (非树边), 318, 320
 NP, 595, 596
 NP-completeness (NP完全性), 592-642
 NP-hard (NP困难的), 600
 null string (空串), 419
 number theory (数论), 453-471

O

O'Rourke, 590
 object-oriented design (面向对象设计), 137
 objective function (目标函数), 259
 Ofman, 284
 one-time pad (一次一密), 474
 one-way hash function (单向散列函数), 481
 online algorithm (在线算法), 667-679
 open addressing (开放定址法), 123, 124
 optimization problem (优化问题), 278, 594
 order statistic (顺序统计量), 245
 orientation (方位), 574, 575
 origin (源点), 290
 Orlin, 338, 379, 414
 orthogonal segment (正交线段), 565
 Ottmann, 590
 out-degree (出度), 290
 outgoing edge (出边), 290
 output sensitive (输出敏感的), 565, 579
 outside node (外结点), 551
 overflow (溢出), 164, 653
 Overmars, 683

P

P, 595
 Pach, 590
 palindrome (回文), 449
 Pan, 510
 Papadimitriou, 284, 642, 683

- parallel algorithm (并行算法), 657-666
- parallel prefix (并行前缀), 662
- parent (父结点), 75
- PARTITION, 640
- partition (划分), 227-234
- partition tree (划分树), 561
- password (口令), 472
- Patashnik, 54
- path (路径), 292
 - directed (有向的), 292
 - length (长度), 341
 - simple (简单的), 292
 - weight (权值), 341
- path compression (路径压缩), 230
- path length (路径长度), 134
- pattern matching (模式匹配), 419-428
 - Boyer-Moore algorithm (Boyer-Moore算法), 422-425
 - brute force (蛮力法), 420-422
 - Knuth-Morris-Pratt algorithm (KMP算法), 425-428
- Patterson, 683
- Peleg, 544
- plane sweep (平面扫描), 566-571, 590
- point (点), 572
- polygon (多边形), 572, 573, 576
 - convex (凸多边形), 573
- edges (边), 573
- non-intersecting (不相交的多边形), 573
- simple (简单多边形), 573
- vertices (顶点), 573
- polynomial (多项式), 52
- polynomial-time approximation scheme (多项式时间的近似模式), 619
- polynomial-time reducible (多项式时间可归约的), 600
- position (位置), 68, 77, 196
- potential function (势能函数), 37-38
- power test (幂测试), 46
- PRAM model (PRAM模型), 657
 - CRCW (并发读并发写), 658
 - CREW (并发读互斥写), 658
 - EREW (互斥读互斥写), 658
- Pratt, 450
- prefix (前缀), 419
- prefix averages (前缀平均值), 31
- prefix code (前缀码), 440
- prefix sum (前缀和), 33
- Preparata, 590
- Prim, 379
- Prim-Jarnik algorithm (Prim-Jarnik算法), 366-367
- primality testing (素性测试), 466-471
 - Rabin-Miller algorithm (Rabin-Miller算法), 470
 - Slovay-Strassen algorithm (Slovay-Strassen 算法), 467-469
- prime (素数), 453
- prime decomposition (素数分解), 453
- primitive operation (基本操作), 9-12, 36
- primitive root (原根), 462
- primitive root of unity (本原单位根), 489
- priority queue (优先队列), 94-113, 218
 - heap implementation (堆实现), 100-106
 - sequence implementation (序列实现), 96-98
- priority range tree (优先范围树), 560-561
- priority search tree (优先查找树), 556-559
- probability (概率), 28-30
- probability space (概率空间), 28
- program counter (程序计数器), 60
- proximity (接近程度), 568
- prune-and-search (剪枝-查找), 245-247
- pseudo-blocks (伪块), 656
- pseudo-code (伪代码), 7-8
- pseudo-polynomial time (伪多项式时间), 281
- pseudo-random number generators (伪随机数生成器), 195
- PTAS, *see* polynomial-time approximation scheme (PTAS, 见多项式时间的近似模式)
- public-key cryptography (公钥密码学), 472, 475-480
- public-key cryptosystem (公钥密码系统), 475
- Pugh, 216
- ## Q
- quadratic probing (二次探测法), 124
- quadratic residue (二次剩余), 467
- quadtree (四叉树), 561-562
- queue (队列), 61-64
 - abstract data type (抽象数据类型), 61
 - array implementation (数组实现), 61-63
- quick-sort (快速排序), 234-238
 - tree (树), 235
- ## R
- Rabin-Miller algorithm (Rabin-Miller算法), 470
- radix-sort (基数排序), 242-243
- Raghavan, 216, 256, 683
- RAM, *see* random-access machine (RAM, 见随机存取机)
- Ramachandran, 137
- random variable (随机变量), 29
- random-access machine (随机存取机), 9, 644
- randomization (随机化), 195, 196
- randomized quick-select (随机化快速选择), 245
- randomized quick-sort (随机化快速排序), 237
- randomized thresholding (随机化门限), 675
- range searching (范围查找)
 - one-dimensional (一维), 550-552

three-sided (三方的), 556
 range tree (范围树), 549-561
 range-search query (范围查找查询), 549
 rank (位序), 142, 231
 rank groups (位序组), 232
 Rao, 510
 ratio test (比率测试), 45
 reachability (可达性), 316
 recurrence equation (递归方程), 12, 224, 263-270
 recursion (递归), 12, 60
 recursion tree (递归树), 265
 recursive doubling (递归倍增), 661
 red-black tree (红黑树), 170-184, 209, 569
 depth property (深度性质), 170
 external property (外部性质), 170
 internal property (内部性质), 170
 recoloring (重着色), 174
 root property (根性质), 170
 Reed, 137
 reflexive property (反射性质), 94, 308
 rehashing (再散列法), 122
 Reif, 683
 relatively prime (互素), 454, 458
 relaxation (松弛), 343, 349
 renter's dilemma (租赁人难题), 667
 repeated squaring (反复平方), 463
 rescalable (可伸缩的), 619
 residual capacity (剩余容量), 387
 residual distance (剩余距离), 393
 residual graph (剩余图), 392
 residue (剩余), 458
 restriction (限制法), 603
 restructure (重构)
 trinode (三结点), 155
 Ribeiro-Neto, 450
 right child (右子结点), 76
 right subtree (右子树), 76
 right turn (右旋转), 574
 Rivest, 216, 338, 379, 414, 476, 510, 642
 Robson, 137
 root (根), 75
 root object (根对象), 323
 Rosenkrantz, 642
 rotation (旋转), 155, 157
 double (双), 155
 single (单), 155
 round-robin (循环), 64
 router (路由器), 530
 routing (路由)
 broadcast (广播), 530-531
 center-based tree (中心树), 537-538

distance vector algorithm (距离矢量算法), 532-533
 link-state algorithm (链路-状态算法), 534-535
 multicast (多播), 535-540
 reverse path forwarding (逆向路径转发), 535-536
 unicast (单播), 532-535
 RPF, *see* reverse path forwarding (RPF, 逆向路径转发)
 RSA, 476
 RSA cryptosystem (RSA密码系统), 476-478
 running time (运行时间), 4-6, 10-12

S

Samet, 590, 683
 sample space (样本空间), 28
 SAT, 638
 satisfying assignment (满足指派), 598
 Saxe, 284
 scan forward (前向扫描), 197
 Schaffer, 137
 Schönhage, 284
 search engine (搜索引擎), 225, 439
 secondary clustering (二次聚集), 124
 security algorithm (安全算法), 481-487
 Sedgewick, 54, 137, 216, 510, 590
 seed (种子), 195
 segment (线段), 572, 573
 segment intersection (线段相交), 565-568
 Seidel, 590
 selection (选择), 245-247
 selection-sort (选择排序), 97, 218
 self-loop (自循环), 291
 sentinel (观察哨), 70, 115
 separate chaining (分离链地址法), 121
 separation edge (孤立边), 307
 separation vertex (孤立顶点), 307
 sequence (序列), 73-74
 abstract data type (抽象数据类型), 73-74
 sequential subset (连续子集), 659
 SET, 484
 set (集合), 225-234
 SET-COVER, 610, 611, 625, 626, 638, 642
 Shamir, 476
 Shamos, 590, 683
 Shiloach, 683
 Shing, 284
 shortest path (最短路径), 341-359
 Bellman-Ford algorithm (Bellman-Ford算法), 349-351
 Dijkstra's algorithm (Dijkstra算法), 342-348, 373-376
 matrix multiplication (矩阵相乘), 355-359
 sibling (兄弟结点), 75
 sieve algorithm (筛选算法), 136

- sink (汇点), 382, 383
 - skip list (跳跃表), 195-202
 - analysis (分析), 200-202
 - insertion (插入), 198-199
 - level (层), 196
 - removal (删除), 199-200
 - searching (查找), 197-198
 - tower (塔), 196
 - update operation (更新操作), 198-200
 - Slcator, 216, 683
 - Solovay-Strassen algorithm (Solovay-Strassen算法), 467
 - sorting (排序), 96, 218-224, 235-243
 - bucket-sort (桶排序), 241-242
 - external-memory (外存), 654-656
 - heap-sort (堆排序), 107-111
 - in-place (原位), 108, 248
 - insertion-sort (插入排序), 98
 - lower bound (下界), 239-240
 - merge-sort (归并排序), 219-224
 - priority-queue (优先队列), 96
 - quick-sort (快速排序), 235-238
 - radix-sort (基数排序), 242-243
 - selection-sort (选择排序), 97
 - stable (稳定的), 242
 - source (源点), 382, 383
 - space usage (空间利用), 4
 - spanning forest (生成森林), 306, 315
 - spanning subgraph (生成子图), 292
 - spanning tree (生成树), 293, 303, 305, 306, 313, 315, 316, 360
 - splay tree (伸展树), 185-194
 - split (分裂), 164, 653
 - stable (稳定的), 242
 - stack (栈), 57-60
 - abstract data type (抽象数据类型), 57
 - array implementation (数组实现), 57-58
 - start time (开始时间), 261
 - Stearns, 642
 - Steiglitz, 284, 642
 - Steiner tree (Steiner树), 538-540
 - Stephen, 450
 - Stirling's Approximation (Stirling近似), 686
 - STL, 137
 - stop word (停止字), 431, 450
 - Strassen, 284
 - Strassen's Algorithm (Strassen 算法), 272
 - string (串)
 - abstract data type (抽象数据类型), 419-420
 - null (空), 419
 - pattern matching, *see* pattern matching (模式匹配, 见模式匹配)
 - prefix (前缀), 419
 - suffix (后缀), 419
 - strongly collision-resistant (强抗冲突的), 481
 - strongly connected (强连通的), 316
 - strongly *NP*-hard (强 NP 困难的), 612
 - subgraph (子图), 292
 - SUBGRAPH-ISOMORPHISM (子图同构), 640
 - subproblem optimality (子问题优化), 275, 278
 - subproblem overlap (子问题重叠), 278
 - subsequence (子序列), 443
 - SUBSET-SUM (子集和), 612-614, 630, 638, 642
 - substring (子串), 419
 - subtree (子树), 75
 - suffix (后缀), 419
 - summation (求和), 21, 687
 - geometric (几何), 21
 - symmetric encryption (对称密码), 473-474
 - symmetric property (对称性质), 308
 - symmetric relation (对称关系), 289
 - Szemerédi, 683
- ## T
- Tamassia, 137, 338, 510
 - Tarjan, 54, 137, 216, 256, 338, 379, 414, 683
 - task scheduling (任务调度), 261
 - Tel, 544
 - telescoping sum (叠缩和), 35, 687
 - template method pattern (模版方法模式), 332
 - text compression (文本压缩), 440-441
 - Theseus, 288
 - thread (线程), 63-64
 - timestamping (时间戳), 482
 - topological ordering (拓扑排序), 325-327
 - total order (全序), 94
 - totient function (欧拉 ϕ 函数), 460
 - tower (塔), 198
 - tower-of-twos (2的幂), 231
 - trailer (尾结点), 70
 - transfer (转移), 167
 - transitive closure (传递闭包), 316, 319
 - transitive property (传递性质), 94, 308
 - traveling salesman problem (旅行推销员问题), 342
 - traveling salesperson problem (旅行推销员问题), 617
 - treap, 590
 - tree (树), 75-137, 292
 - abstract data type (抽象数据类型), 77-78
 - binary, *see* binary tree (二叉, 见二叉树)
 - child node (子结点), 75
 - depth (深度), 79-80
 - external node (外部结点), 75

height (高度), 79-80
 internal node (内部结点), 75
 level (层), 84
 linked structure (链表结构), 93
 multi-way (多路), 159-162
 multidimensional (多维), 549-564
 node (结点), 75
 ordered (有序的), 76
 parent node (父结点), 75
 root node (根结点), 75
 tree edge (树边), 318, 320
 tree traversal (树的遍历), 81-83, 86-89
 Euler tour (欧拉路径), 88-89
 generic (泛型), 88-89
 inorder (中序), 87
 level order (层序), 134
 postorder (后序), 82-83, 86
 preorder (前序), 81, 86
 triangle inequality (三角不等式), 623
 triangulation (三角剖分), 284
 trie, 429-439
 compressed (压缩的), 433
 standard (标准), 429
 trinode restructuring (三结点重构), 155, 173
 Tsakalidis, 216
 TSP, 617, 619, 623, 624, 633, 641, 642
 Tukey, 510
 two-dimensional dictionary (二维字典), 549
 (2,4)tree ((2,4)树), 163-169
 depth property (深度性质), 163
 size property (大小性质), 163

U

Ullman, 137, 216, 256, 510, 642, 683
 unary encoding (一元编码), 593
 underflow (下溢), 167, 653
 unicast routing (单播路由), 530, 532-535
 union-by-size (按规模合并), 230
 union-find (合并寻找), 227-234
 universal hashing (通用散列), 125-127
 unordered dictionary (无序字典), 114
 up-heap bubbling (堆向上冒泡), 102

upper envelope (上包围), 283

V

Valiant, 683
 van Leeuwen, 338, 379, 414
 vector (向量、矢量), 65-67, 115, 142
 abstract data type (抽象数据类型), 65
 verification (验证), 596
 vertex (顶点), 289
 degree (度), 290
 in-degree (入度), 290
 out-degree (出度), 290
 vertex cover (顶点覆盖), 598
 VERTEX-COVER, 599, 608-613, 615, 622, 642
 virtual memory (虚拟内存), 647
 Vishkin, 137, 683
 Vismara, 590
 Vitter, 54, 683
 Voronoi diagram (Voronoi图), 589
 Vuillemin, 590

W

Web crawler (Web爬虫), 303, 443
 Web spider (Web蜘蛛), 303, 443
 Wegner, 137
 Williams, 137
 Wood, 137, 590

X

X.590, 486

Y

Yannakakis, 642
 Yao, 590
 Yap, 510
 Yellin, 137
 Young, 642

Z

zig, 186, 192
 zig-zag, 186, 192
 zig-zig, 185, 192